1999

A genetic algorithm test bed implementation.

Thomas Albert. Williams
University of Windsor

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

A Genetic Algorithm Test Bed Implementation

by
Tom Williams

A Thesis
Submitted to the College of Graduate Studies & Research
in Partial Fulfillment of the Requirements for the Degree of
Master of Science at the University of Windsor
Windsor, Ontario,
Canada
1998

888414

Abstract

Genetic algorithms are a powerful tool for solving search and optimization problems. We examine the problems associated with applying a genetic algorithm and develop a new computer program to facilitate the rapid application of genetic algorithms. This system is intended to function as a test bed for GA research or as a general purpose solution mechanism.

Our approach is to place all of the common non-problem specific logic in a menu driven testbed program. The problem specific objective function is created separately by the user and is treated as an external program. The testbed program is capable of interfacing with the external objective function whenever necessary

Maximum flexibility in applying genetic algorithms is obtained by making the testbed program platform independent and by allowing the external objective function to be created by any programming tool available for the desired platform.

Using these design goals we have developed an extremely flexible and easy to use tool for applying genetic algorithms to any desired problem area.

To Joanne

Acknowledgments

I would like to acknowledge the guidance and support provided by Dr. Subir Bandyopadhyay. His advice, suggestions and comments have been invaluable. I would also like to thank Ziad Kobti and Jim Smith for there ever present ideas and discussions.

Table of Contents

List of Figures

1. Introduction

1.1. Our Area of Research

In this thesis we are interested in developing a complete system for rapidly and easily applying genetic algorithm technology to any problem.

Genetic algorithms (GAs) are used for finding solutions to search and optimization problems using computational methods that are based on the observed behavior of biological systems with respect to evolution. They mimic the properties of mating to provide for an artificial evolution of solutions within a given problem domain.

1.2. The Problems Addressed in this Thesis

The use of genetic algorithms for solving a particular problem is not trivial. It usually involves utilizing some form of GA function library or a GA tool kit. Either of these requires the user to do a great deal of computer programming.

These tool kits or libraries require the user either be proficient in certain computer languages or to become so, and assume that the user has a particular type of computer system on which to run the system.

1.3. Our Solution

Our solution is to provide a complete genetic algorithm testbed program which provides all of the non problem specific logic and has provisions for performing the algorithm in most of its commonly used variations.

Computer programming skills are kept to an absolute minimum by requiring only that the user supply a problem specific objective or fitness function and optionally a user supplied function to help interpret the results. To make this even easier, the user has complete freedom to choose any programming tool desired to do this.

1.4. Thesis Outline

After the introduction in Chapter 1, a background discussion on genetic algorithms is presented in Chapter 2. This starts by defining the genetic algorithm and comparing it with other classes of search techniques. We then look at the operation of the genetic algorithm in greater detail, including the three operators of the genetic algorithm cycle: selection, reproduction and crossover. Some of the modern advancements and refinements to the basic algorithms are also presented.

The use of genetic algorithms to solve problems is discussed next in Chapter 3. This looks at the suitability of using a genetic algorithm on a particular problem and discusses details such as good chromosome coding and handling problem constraints with their resultant lethal chromosomes. It also looks at problems associated with developing a genetic algorithm program from scratch and by using publicly available GA tool kits.

The features of the genetic algorithm testbed are discussed next in Chapter 4. This includes a discussion of the flexibility and completeness of the system as well as the platform independent features and of using it for rapid deployment in a familiar computing environment. It also outlines the usage of the testbed by indicating the parameters available for selection from the menu.

The details of the implementation of the GA testbed are discussed in Chapter 5. This includes a discussion of the Java program used for the system as well as the Java native methods used to interface to user supplied external functions.

The details of the methodology used for verifying the accuracy of the testbed and its results are discussed in Chapter 6. This includes a discussion of the methods used for assuring the proper functioning of all the

individual components used to create the system as well as the overall testing done using several example problems.

In Chapter 7, the limitations of the current testbed implementation are discussed and possible future modifications and enhancement are outlined.

Appendix A is a list of abbreviations. Appendix B is a set of definitions for the subject area. Appendix C is a survey of genetic algorithm research and Appendix D is a bibliography of genetic algorithms. Appendix E is the source code for the survey Appendix F is the source code for the GA testbed implementation and Appendix G is the source code for the example problems used to verify the GA testbed.

2. Background

2.1. Introduction

Genetic algorithms (GAs) are an important class of computational methods that are used to find solutions to difficult search and optimization problems. This solution mechanism is borrowed from nature and is based on natural evolutionary genetics. They have been successfully utilized in many problem areas including game playing, pattern recognition and structural design.

The GA is robust since the only requirement for applying it to a particular problem is that the solution can be coded into a character string called a chromosome and that there is some method to determine the quality or *fitness* of each solution. No other information about the problem is needed. This means that a GA can be applied to a wide variety of problems including some where there is no other practical solution technique.

This GA background chapter provides the information necessary for understanding the GA testbed program. A more complete GA discussion is included in the survey in Appendix C.

2.2. What is a Genetic Algorithm?

A genetic algorithm is a computer based search and optimization technique first introduced by John Holland and reported in his 1975 book *Adaptation in Natural and Artificial Systems* [HOL75]. He took the idea first defined by Charles Darwin in *The Origin of Species,* that in nature, populations evolve according to the principles of natural selection and survival of the fittest. He then applied it to computer based problem solving. In a manner similar to that of nature, genetic algorithms use evolution to solve real world problems. They have found use in such diverse areas as design, on-line process control, and load balancing.

Genetic algorithms emulate the main processes involved in natural evolution of biological species. In any species or population there is competition for valuable resources such as food, water and shelter. In this situation those individuals that are most suited for this competition will thrive and reproduce. During reproduction those character traits that helped this individual be more adapted to its environment will be passed to the next generation. Over time and many generations, the most desirable traits will tend to be concentrated in the population while the least desirable traits will

greatly diminish. In this manner, populations tend to adapt to their environment over time.

This adaptation principle can be applied to computer problem solving by identifying the important facets of evolution and emulating them in a computer algorithm. Genetic algorithms use a direct analogy of evolution through natural selection. In a GA, each individual represents a solution to a given problem. The individual solutions are ranked according to their ability to solve the given problem. Individuals are chosen for survival and consequent mating using a biased random selection technique. It is biased such that the chance of an individual being selected is directly proportional to it's fitness.

Each surviving individual is randomly matched to another surviving individual. These two become the parents in a mating process which involves creating two new offspring chromosomes. This mating is accomplished by mixing elements of each of the parent chromosomes to produce offspring representing completely new solutions to the given problem. If the parents were both fairly fit then there is a chance that an offspring could inherit the best of each parent and surpass either parent in fitness.

By performing this mating process on the entire population of solution individuals a whole new population of offspring is created. This new generation will likely contain a higher proportion of those characteristics that are best suited for solving the problem. The same selection and mating procedure can now be applied to this new generation to create a third generation. With each succeeding generation there will be an increase in the overall fitness of these individuals in solving the problem.

While GAs are not guaranteed to find the global optimum, they are good at finding good solutions in a reasonable amount of time. However, when GAs are compared to specialized techniques for solving particular problems, they usually perform poorly in comparison. In general, any custom designed solution technique will outperform a general solution technique such as a GA. The power of the GA is that it produces good results without having to be custom designed for each problem domain.

2.3. Classes of Search Techniques

There are a great number of search and optimization techniques, some of which are general purpose like the genetic algorithm, and others which are designed for a specific problem domain [Gold89a] such as

dynamic programming. Some of the more popular search techniques are random search, gradient methods, iterative search and simulated annealing.

2.3.1. Random Search

Random search is a brute force approach, usually used for difficult functions where there are no other viable search methods. In this approach, points in the search space are selected randomly and their fitnesses are evaluated. This strategy is not very intelligent and is rarely used alone. It is a little like searching for a needle in a hay stack because there is no guidance to direct the search.

2.3.2. Gradient Methods

There are a number of methods for optimizing continuous functions, all of which are based on using information about the gradient of the function to guide the direction of search. A drawback is that these methods will only work on problems where the derivative of the function is continuous.

These methods are generally referred to as hill climbing methods since they usually move upwards along a gradient in the hopes of finding a maxima. On functions that are multi-modal, where there are many peaks,

these gradient techniques can be trapped in a peak that is perhaps the highest in the local vicinity but not the highest globally. Once they have reached the top of a local maximum no further progress can be made towards a solution.

2.3.3. Iterated Search

When random search and gradient search are combined they give an iterated hill climbing search. In this method, the gradient method is used to find a peak but once a peak has been located the hill climbing is started over again, this time using a different randomly chosen starting point. This gives this technique the advantage of simplicity and gives it the ability to perform well even if the function has only a few local maxima.

This method is nothing more than several iterations of the gradient method and it cannot find an overall picture of the shape of the domain. Therefore, as the search progresses it cannot use any previously obtained information to choose a likely starting point for the next hill climb. This forces it to evaluate points in regions of low fitness, just as often as in regions found to be of high fitness.

A genetic algorithm, by comparison, starts with an initial randomized population but allocates increasing trials to regions of the search space found to have higher fitness.

2.3.4. Simulated Annealing

Simulated annealing is a process that is analogous to the cooling of a metal from a molten state. In this method, an initial search point is chosen at random. A move is made to another point in the search space also chosen at random. If the move is to a point with higher fitness then it is accepted. However, if the move is to a point with lower fitness its acceptance is determined by a probability function which varies over time. This function begins with a value near one but gradually reduces towards zero.

In this way, the search can proceed up and down in the beginning but as the search starts to "cool" the amount of downward mobility is reduced. Towards the end of the search, only upward moves are allowed. Downward moves are essential if local maxima are to be escaped but after a period of time it is hoped that the search will be near the peak of the global maximum and the cooling temperature will force the search to proceed upwards towards this maximum.

This technique only deals with one solution at a time and does not build up an overall picture of the search space. Information from previous moves is not used in the selection of the new moves. This is in direct contrast to a genetic algorithm where the search space peaks are embedded in the chromosomes of the population individuals and tend to be passed forward during the search.

2.4. The Genetic Algorithm Mechanism

As a biologically inspired search method, the GA mechanism copies the essential operations of natural genetics. It operates on a large group of solutions known as a population and simulates the biological operations of survival, mating and mutation. As in the biological cycle of birth, reproduction and death, a GA follows a similar cycle called the *GA cycle* [Gol89a].

2.4.1. Solution Coding: The Chromosome

Similar to a biological chromosome, a GA chromosome is a string of characters that represent the instructions for "building" an individual. Since, in the GA realm, an individual is a solution to a problem, a GA chromosome

must contain the coded instructions for "building" a solution. That is, the GA chromosome is the solution in a coded format.

In biology, a chromosome is composed of a long string of proteins. Each of the positions along the string is called a gene and can be filled with one of four types of proteins. These protein types are the gene values and are called alleles.

In a GA chromosome, each position along the string is called a gene but characters are used for the gene values (alleles) instead of proteins. So, unlike biology, we are not limited to choosing from four values represented by the four available proteins. Instead, we can choose to draw our characters from any fixed alphabet.

In practice, the freedom to choose characters from any fixed alphabet is not all that useful. This is because Holland [Hol75] has shown that a longer chromosome string is superior to a shorter one, which means that an alphabet with fewer characters is superior to one with more. Since the smallest usable alphabet is one with just 2 characters, an alphabet composed of the characters 0 and 1 is often employed. Chromosomes made from this alphabet will form binary number strings which are easy to

manipulate. It is said that such a chromosome has an alphabet cardinality of 2.

2.4.2. The GA Cycle

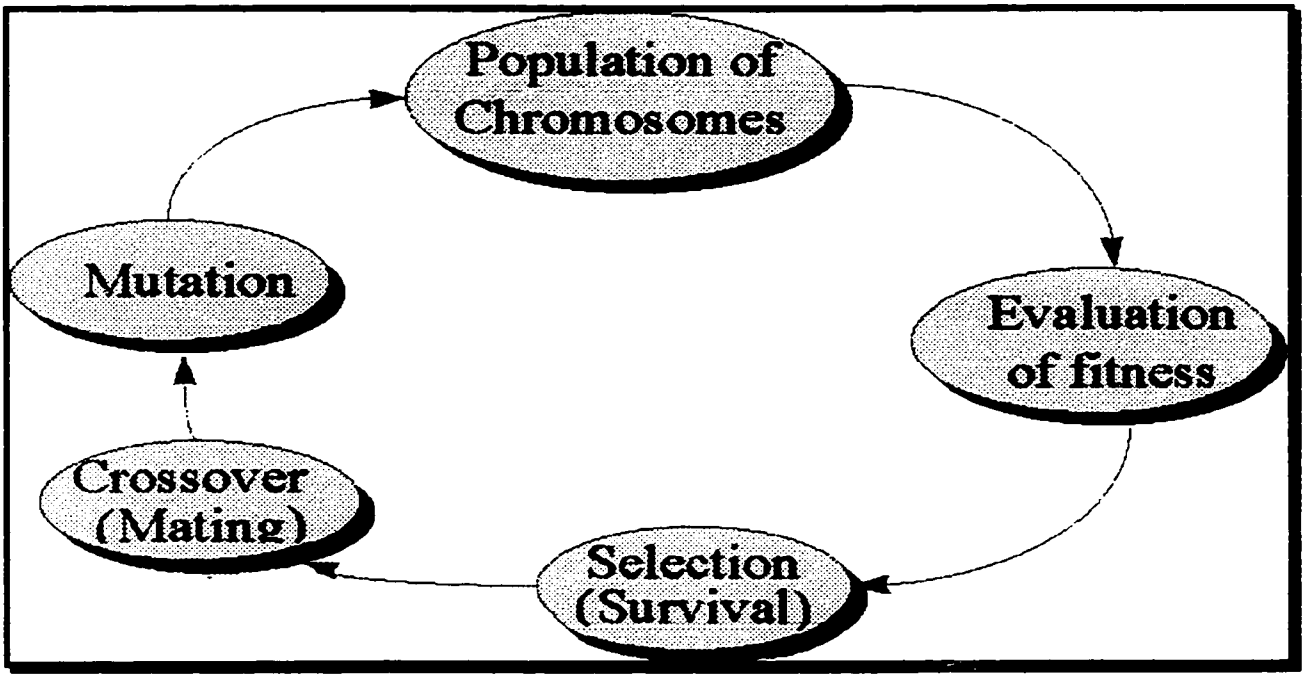

Figure 1 The GA Cycle

The GA cycle starts with a population of chromosome strings that are chosen randomly, each of which represents a solution to the problem. Each member of the population is subjected to the operations of evaluation, selection, crossover and mutation as shown in Figure 1. The result of this is a population of new members. The GA cycle is repeated until either the

desired fitness is obtained or until fitness improvements diminish. In this situation the GA is said the to have converged.

The GA cycle operators are detailed in the following sections.

2.4.3. Fitness Evaluation

In order to evaluate the effectiveness of a solution there must be an objective or fitness function that takes a solution chromosome and determines a figure of merit for this solution. A higher figure of merit indicates a superior or fitter solution.

2.4.4. Selection (Survival)

Selection is the process of choosing chromosome strings for mating. A chromosome string with a higher fitness value will have a higher probability of being chosen. This is analogous to the natural selection process whereby an organism that is more fit has a higher chance of survival.

The new population that results from the selection operator is biased towards those individuals that are the fittest. The population of selected individuals is next subjected to crossover.

The selection process may utilize the elitism operation where the most fit individual is always selected for inclusion in the next generation.

2.4.5. Crossover (Mating)

There are several methods for performing crossover. In each case a new chromosome is created when one or more substrings from the chromosome of one parent is combined with one or more substrings from the chromosome of the other parent, to form a whole new string. In this way, genetic information from two individuals is transferred to a single offspring.

The substrings from each parent that were not used to produce the first offspring are combined to produces a second offspring. With each pair producing two offspring, the population size remains constant.

2.4.5.1. One Point Crossover

The GA performs 1 point crossover by making a single cut at the same location in each of the two parent chromosomes. The cut sections are then exchanged to form two offspring.

For example, suppose we let the first parent be an arbitrary 7-bit binary string represented by the sequence ABCDEFG where A represents the most significant bit and G the least significant bit and similarly we let the

second parent be represented by abcdefg. They would mate using 1-point crossover as follows:

1. A random cut point between 1 and 6 is chosen (e.g. 3)
2. Get substring1 from the first parent between the start and the cut point
3. Get substring from the second parent between the cut point to the end
4. The resulting offspring is a combination of the two substrings from steps 2 & 3 as shown in Figure 2
5. Reverse parents and repeat steps 2, 3, & 4 yielding the second offspring (abcDEFG)

```
First parent = ABC|DEFG          substring1 = ABC

Second parent = abc|defg         substring2 = defg

offspring = substring1 + substring2  =  ABCdefg
```

Figure 2 One Point Crossover

2.4.5.2. Two Point Crossover

In 2-point crossover, chromosomes are arranged in loops by joining their ends together. Two cuts are made in the loop and the resulting

segments are exchanged. The offspring gets the genes between the cut point 1 and cut point 2 from the first parent and all the rest from the second.

In Figure 3 the chromosome of one parent is shown in this arrangement with each gene represented by the symbol ⊗. The chromosome begins next to the start-finish line and goes counter-clockwise in a circle back to the start-finish line. Two arbitrary cut points are shown, one between genes 4 & 5 and the other between genes 5 & 6. In this case, the segment between the two cut points is just one gene: number 5. So the offspring would get gene 5 from the first parent and all the rest from the second parent.

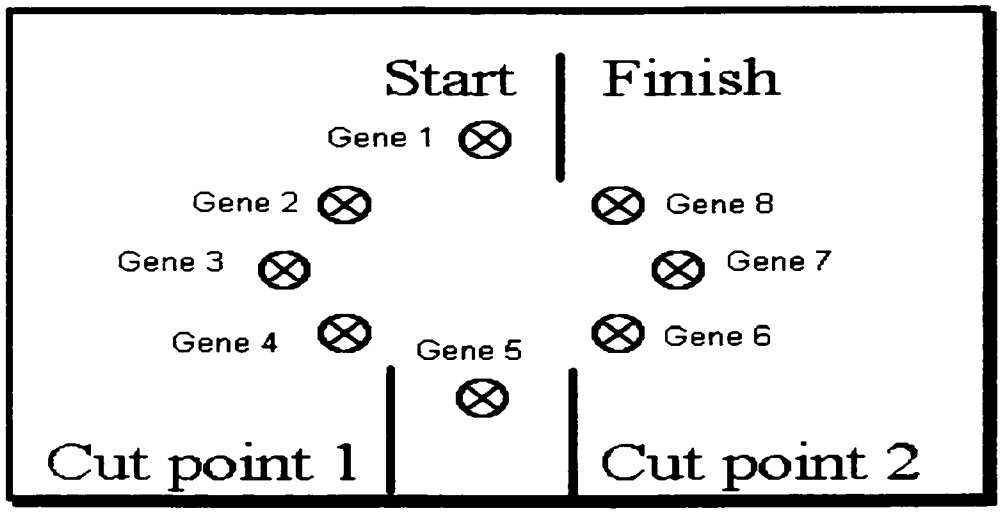

Figure 3 Two Point Crossover.

From this it can be seen that 1-point crossover is just a special case of the more general 2-point crossover where one of the cut points is fixed at the start - finish line. This would account for the increased performance seen when 2-point crossover is used. It is no more disruptive than 1-point crossover since they both have 2 cut points and 2-point crossover does not always destroy building blocks with widely spaced genes as is the case for 1-point crossover.

It is generally considered that 2-point crossover is superior to 1-point crossover [DeJ75].

2.4.5.3. Uniform Crossover

Another form of crossover is the N-point uniform crossover where the number of points N varies dynamically with each mating. In this method, a randomly generated crossover mask is used to determine which genes of an offspring come from which parent. Each gene in the first offspring is created by copying the corresponding gene from one or the other parents according to the crossover mask. Where there is a 1 in the mask, the gene is copied from the first parent, and where there is a 0 in the mask, the gene is copied from the second parent. The process is repeated with the parents exchanged to produce the second offspring. A new crossover mask is

randomly generated for each pair of parents. Offspring, therefore, contain a mixture of genes from each parent. The number of effective crossing points, while not fixed, will average L/2 (where L is the length of the chromosome).

For example, suppose we let the first parent be an arbitrary 7-bit binary string represented by the sequence ABCDEFG where A represents the most significant bit and G the least significant bit and similarly we let the second parent be represented by abcdefg. They would mate using uniform crossover as follows:

1. A random crossover mask is chosen (e.g. 0011101001)
2. Wherever the mask has a 1 the corresponding bit from the first parent is chosen
3. Wherever the mask has a 0 the corresponding bit from the second parent is chosen
4. The resulting offspring is combined with bits as chosen in steps 2 & 3 as shown in Figure 4.
5. Reverse parents and repeat steps 2, 3, & 4 yielding the second offspring A B c d e F g

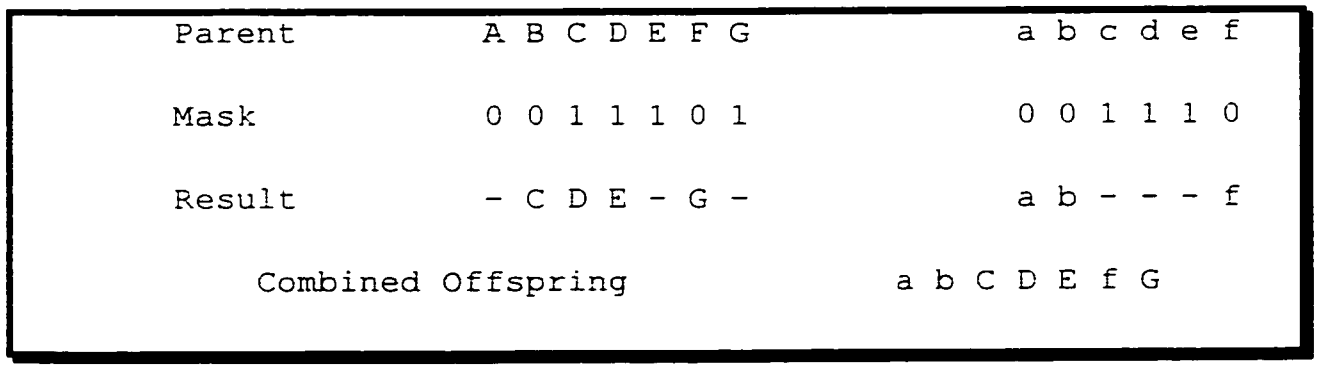

Figure 4 Uniform Crossover

2.4.6. Mutation

Mutation is the process whereby a randomly chosen gene is changed to a randomly chosen value. Normally, mutation occurs with low probability and functions as a background operator [Boo87][DeJ85]. It is included to enable the GA to search space that may otherwise be precluded by the converging chromosomes as genetic information is discarded during crossover. The exact amount of mutation necessary is somewhat open to debate. Too little and useful alleles that are not currently in the population can never be found while too much causes the GA to degenerate into a random search.

2.5. Conclusion

Genetic algorithms are a robust technique for solving difficult search and optimization problems. The only problem domain information that they require is the fitness of a potential solution. This allows them to be used where other methods fail. Nature has shown the way to a technique is simple yet powerful.

3. Problem Solving With Genetic Algorithms

3.1. Introduction

Solving a problem with a genetic algorithm requires three major undertakings. The first undertaking is to devise a good chromosome coding scheme that adequately represents the solution while avoiding some common coding problems. The success or failure of a GA with regard to particular problem can be directly related to the coding used for the chromosome string [VL91].

The second major undertaking is to program the objective function. Since all problems are different they will require a unique objective function. This is usually not too difficult if the problem is well understood.

Beyond this, there is also the requirement to implement the GA cycle operators. Mechanisms are needed to generate a large population of solutions as well as to perform the selection, crossover and mutation operations.

There are practical aspects to each of these undertakings. The successful application of a GA requires that these conditions be considered

and addressed. With some care and understanding many of the common pitfalls can be avoided.

3.2. GA Suitability

Not all problems are suitable for genetic algorithms. Some problems, termed GA deceptive, exhibit peculiar characteristics [Gol89a], [Gre93].

Under normal circumstances, those genetic building blocks which are contained in the global optimum will increase in frequency throughout the population. Eventually, these optimal genetic building blocks will come together in a single individual and the globally optimum chromosome will be constructed. In a GA deceptive problem though, the genetic building blocks which are not contained in the global optimum increase in frequency more rapidly than those which are. In this case, the GA will be misled away from the global optimum, instead of towards it. This phenomenon is directly related to the detrimental effects of epistasis [Gol87] [Gol89a][DG91] in a GA.

3.2.1. Epistasis

Epistasis is the name for those interactions which occur between the problem parameters. In bridge building for example, the length of a

suspension cable is related to the length of all the beams making up the span as well as the height of the towers above the span. So beam lengths, span height and cable length are inter-related in some non-linear fashion. A change in one parameter will affect the others.

In a GA, when inter-related parameters such as these are coded as genes, the genes are said to exhibit epistasis. If each gene can be treated as representing an independent variable, there will be no epistasis. Such a problem would be much easier to solve because each parameter being independent could be solved in isolation from the other variables. Since most problems which require a genetic algorithm are non linear, there is usually some degree of epistasis.

The problems associated with epistasis may be lessened by using a different coding scheme which does not exhibit epistasis. It has been shown that [VL91] in principle, any problem can be coded in such a way as to make it simple. However, the effort involved in doing so will probably be considerable, and could effectively constitute solving the initial problem.

If the chromosome coding scheme used exhibits too much epistasis between its genes then the GA will be deceived and the population will not

converge toward superior solutions. A GA is not suitable for use on such a problem.

3.2.2. Testing a Problem for GA Suitability

In most cases, it is not possible to determine in advance if a particular class of problems with a particular chromosome coding scheme will be GA deceptive [Gol89a]. It is often necessary to actually run the problem to see if the population is converging.

Sometimes it may be possible to test using a simple problem from the same class of problems as the more difficult problem that you are actually interested in. If a GA is useful for one type within some class of problems it is usually useful for all types within that class.

3.3. Good Chromosome Coding

Perhaps the trickiest part of this whole exercise is to come up with a good coding scheme for the problem solutions. In any given problem there are usually several ways to code the solution as an alphabetic string. A good coding scheme, as noted earlier, will have the longest possible chromosome. As well, it will have as few as possible lethal values where the chromosome does not actually represent a viable solution to the problem.

Even when these guidelines are followed one coding scheme can still be better than another for a given problem. This is due partly to the effects of epistasis discussed above and partly due to the ordering of the genes and the probability that a particular sequence will be split during crossover. There really is no way in advance to determine a good gene ordering other than to test it using an actual GA.

3.3.1. Lethal Chromosomes

In most chromosome coding schemes, due to constraints on the problem solutions, there will be some chromosome values that do not correspond to a viable problem solution. These values are called lethals since an individual with such a chromosome is so poorly fit that it can not survive within the problem environment. Lethals are generally handled by assigning such a chromosome a very poor fitness value [Gol89a]. Since individuals are chosen for mating at random, with a bias towards the most fit individuals, a lethal chromosome will likely not be chosen for mating.

If the population has an excessive number of lethal chromosomes then the GA search can degenerate into a random search. It is best if the

chromosome coding is such that the possibility of creating a lethal solution is minimal.

3.4. Choosing a Chromosome Alphabet

As noted previously, It is generally considered best to use a binary coded chromosome, but there are situations where it may be more convenient to use some other alphabet. Binary is considered best because it creates the longest strings providing more cut points where genetic material can be interchanged during crossover.

In certain situations, a problem may be structured such that it naturally tends to fit a coding in some other alphabet. There is no reason why any alphabet can not be used as long as the problem coding is correct for this alphabet. After all, natural biology seems to do fine with a chromosome alphabet of four.

3.5. Problem Constraints and Invalid Chromosomes

Many problems have a number of constraints giving rise to the unfortunate side effect that certain chromosome values may violate one or more of these constraints. It is also possible that the number of discrete

solutions to the problem is not a power of 2 so it can not be expressed exactly as a binary number.

In this case, the binary number will be larger than the number of available solutions resulting in a number of chromosome values that are not valid. A simple example of this is the problem to find the largest number between 1 and 100. A 6 bit binary chromosome going from 0 to 63 is not large enough to encode this, while a 7 bit chromosome's range of 0 to 127 would have 28 invalid chromosome values.

The usual method of dealing with this is to assign an invalid chromosome a low fitness value. This is justified in that a chromosome that is invalid is not fit to survive and thus should be graded accordingly.

3.6. Objective Function Realization

The objective function is a problem specific function that takes a chromosome and evaluates it to determine how well it solves the problem. It returns a fitness value that is directly proportional to its performance in solving the problem. Creating the objective function is usually straight forward. If however, the problem is not well understood or is a complicated problem, then the realization of an objective function may be more difficult.

3.7. Programming Considerations

Once the problem has been coded as a chromosome string and the objective function has been realized, then the genetic algorithm mechanism can be programmed. It is custom tailored to the number of genes as well as the alphabet used in coding the chromosome string. There are several variations that can be used for the crossover, mutation and selection operators as well as for the GA characteristics such as population size, mutation rate and crossover rate.

This programming can be done either by starting from scratch or by using one of the many available tool kits designed for this purpose.

3.7.1. Hard Coding a GA

Perhaps the most flexible way of programming a genetic algorithm is to create it from scratch. This is not only extremely time consuming but very error prone. It has the advantage of allowing for custom changes to be made to the genetic algorithm mechanism but in many cases it is like reinventing the wheel. The enormous effort involved suggests that there would very seldom be a need to take this route.

3.7.2. GA Toolkits and Library Functions

A genetic algorithm tool kit is much quicker and easier to use than coding from scratch. In such a kit the basic mechanism for doing a genetic algorithm is already programmed and the user merely modifies an existing fitness function with one that evaluates the fitness for the specific problem at hand. Individual genetic algorithm parameters can be set by changing specific constants within the programming toolkit.

This is a fairly flexible way of doing things since it eliminates the need to spend time coding the GA operators which are included in the toolkit. However, the choice of a tool kit is difficult since they are usually written in a specific programming language and all of the custom changes must be made in that language. Therefore, the problem solver must be proficient in using this language on the platform on which the toolkit was designed to run.

3.8. Conclusion

Genetic algorithms are a powerful tool for finding solutions to difficult problems. They are not foolproof and require some understanding of the potential problems that can beset a project. With some forethought and proper design, in most cases these problems are not insurmountable.

4. Genetic Algorithm Testbed Features

4.1. Introduction

The genetic algorithm testbed features a customizable genetic algorithm that includes everything except the problem-specific objective function. It is designed to be much quicker and easier to use than a GA toolkit. Furthermore, there are no computer programming language restrictions placed on the user for creating the objective function.

A genetic algorithm testbed is a platform on which a genetic algorithm can be quickly and easily created while requiring a minimum of programming. This is different from a genetic algorithm toolkit in that there is no need to modify the original program. The GA testbed has a built-in interface to allow it to interact with a custom designed fitness function. The GA testbed provides ease of use and can be used effectively on problems ranging from simple to as complex as they get.

4.2. Rapid Deployment

The GA testbed is designed to allow for a rapid deployment of genetic algorithm technology to a problem area. By encapsulating all of the genetic algorithm mechanisms inside the testbed itself, there is very little

programming to be done. As well, genetic algorithm parameter selection is a simple matter of selecting the correct parameters from a menu.

Rapid deployment makes it possible to use a genetic algorithm to test to see if a given problem will respond with better solutions when subjected to a GA. In the past, it was possible to spend considerable time implementing a GA to solve a specific problem only to find that the problem is GA deceptive and is not readily solvable with a GA.

4.3. Platform Independence

Another feature of the GA testbed is that it is designed to be platform independent. This means that the system can be run on virtually any computer platform available using any operating system available. This can facilitate the sharing of results with colleagues even if they happen to use a different platform. Also, simulations can be run on several different computers at once even if those computers are of a different type. If it turns out that a particular solution is taking too long on one platform, it is simple matter to move the problem to a faster, more powerful computer system.

4.3.1. Familiar Environment

By allowing the user to employ virtually any computer system, the user is not forced to change computer environments. As long as the users regular environment is fast enough for the desired application there will be no need to learn an unfamiliar system.

This is a great advantage over genetic algorithm toolkits which require that the user be familiar with the platform used by the toolkit. This can often entail a steep learning curve for the user, making the process much slower and more error prone.

4.4. Programming Ease

The only portion of the genetic algorithm that must be programmed by the user is the fitness function and optionally a chromosome decoder function. All of the other functions are included in the testbed. GA parameters, such as mutation rate and population size, are not programmed but are instead input by the user at run time. This substantially reduces the amount of programming required when compared to the traditional method of using GA libraries or toolkits.

The familiar computing environment allowed by the use of platform independence makes the required programming much easier. Programming on an unfamiliar system can be time consuming if the learning curve is steep.

4.4.1. No Programming Language Restrictions

In addition to allowing for a familiar programming environment, the GA testbed has an even more powerful feature to ease the programming burden. The GA testbed is designed to interact with user supplied functions that are stand-alone command-line programs.

This frees the user of all programming language restrictions. Any programming language that can create a command line executable program can be used. It does not matter if it is compiled or interpreted. Such languages include C ,C++, Pascal, Basic, Visual Basic, Visual C++, Power Builder , Small Talk and many, many others. Even simple scripting languages such as UNIX Shell or DOS batch programming can be used.

This programming flexibility makes it possible for a person to simply choose whatever programming environment they are most familiar with and not have to learn a new environment. It makes the creation of a fitness

function a very easy process for anyone who has any kind of programming experience.

4.5. Flexibility and Completeness

The GA testbed, while featuring outstanding flexibility is a complete package, has built-in logic for many of the different types of GA features that may be required for solving different problems. This flexibility and completeness makes for an easy to use and powerful search and optimization tool.

4.6. Parameter Selection

The GA testbed console is the user interface which is designed to be easy to use. All of the parameters that are adjustable are displayed in a single window. When a problem is to be solved, the user simply selects the proper settings by typing them into the appropriate spaces. Some of the settings that can be adjusted are the population size, crossover rate, mutation rate, fitness function name, the name of an appropriate chromosome decoder, the number of symbols (cardinallity) in the alphabet used to code the chromosome string and the number of generations to run before quitting.

Figure 5 shows the GA testbed console with the text boxes and drop down menus that are used to enter the parameter selections. The settings shown are the default settings that the GA console starts with. These are the most commonly used setting and can be left alone or changed as desired. Once the parameters have been entered or selected, the button marked "Run New GA" is clicked to start the genetic algorithm.

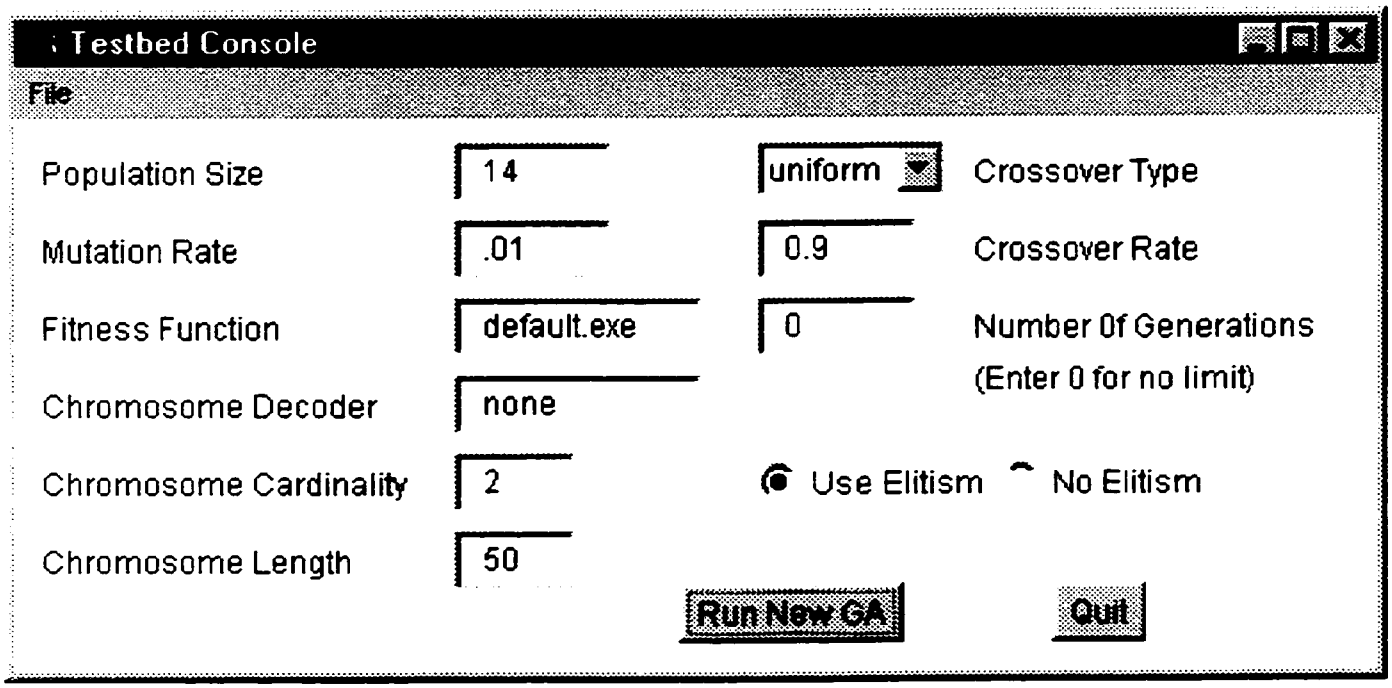

Figure 5 GA Testbed Console

4.7. Conclusion

The GA testbed provides a flexible yet powerful way to solve difficult problems. The system is designed for the maximum amount of freedom while retaining all of the necessary functionality for solving complex problems.

The ease of use and the rapid deployment features make this one of the easiest packages to use for GA research. It is so flexible that it provides a nearly universal platform for doing genetic algorithm work.

5. A Genetic Algorithm Testbed Implementation

5.1. Introduction

The implementation of the GA testbed utilizes several special techniques to accomplish the required features described above. This is especially true for the platform independence and relaxed command-line programming interface features.

5.2. Java Programming

In order to provide for a platform independent package, the software had to be written in a platform independent language. One of the best and most current of these is Java. This language is most noted for creating Web based applets and is not usually associated with a stand alone application such as the GA testbed.

While it is possible to create a stand alone application using Java, there are some special considerations that have to be taken into account to create the interface to the command-line objective function. This connection to the machine programming level is known as Java native methods.

Many programming languages refer to callable code segments such as functions or subroutines. Java though, like most object oriented languages, refers to them as methods. The term *method* is used here when referring to any Java code while the term *function* is used when referring to external code such as the objective function.

5.3. Native Methods

Java native methods are a special interface that allows the interpreted Java code to call a real non-Java native function on the host machine. This is done through a special interface that is written in the C language. It requires that there be a small C language program that runs on the native machine.

In the case of the GA testbed, the small C language program does nothing other than to call the assigned function from the command line. It also contains a small amount of code (which will be discussed later) to retrieve the fitness value from the objective function and transform it into a suitable data type. In this manner, the Java based genetic algorithm can call a command-line fitness or chromosome decoder functions.

While the inclusion of a C language program would seem to negate the main reason for using Java, namely platform independence, it does not. The program is so small, and written using only the most basic of C commands that it can be compiled with virtually any C compiler. This does slightly limit the number of target computer systems to those with a C compiler available.

5.3.1. Command Line Objective Function

In order to give a chromosome to the fitness function, it is passed as a parameter on the command line when the function is activated. The fitness function takes this as an input argument, evaluates its fitness and returns the fitness, value to the Java program.

Unfortunately, the only direct return mechanism from a command line program is an exit code which is an integer between 0 and 255. This range is not sufficient to handle the possible fitness values that will be required from most problems so another indirect mechanism had to be devised. This turned out be a file passing mechanism.

The Java program creates a unique file name and passes this to the objective function on the command line along with the chromosome string. Once the objective function has evaluated the chromosome for fitness, it

then writes that fitness value into a file that is named according to the unique file name that was passed to it.

Upon returning from the fitness function the Java program opens this file and reads the fitness value. It then deletes the file. For example, if the objective function were called MyObjectFunc and it is passed both the chromosome string 01010101010 and a unique filename GA0005 then the following command line would be issued to the system through the Java native method interface:

%MyObjectFunc 01010101010 GA0005

This is executed by the system in exactly the same manner as if it were typed at the command prompt by a user (the command prompt shown, (%) is from UNIX). The user supplied program MyObjectFunc evaluates the chromosome 01010101010 and writes the resulting fitness value to a newly created file called GA0005 which the testbed will read and delete later.

This mechanism of creating and deleting a file for each fitness function call can introduce a great deal of overhead. Disk drives, being mechanical, are slow compared to internal computer system speeds. This does not always cause slowdowns though, since on many machines disk drive accesses are cached. This means that the creation of a file does not

always create a file on the disk, especially, as in the GA testbed case, where the file life is short, it may only ever exist in the memory used for the disk cache avoiding the time penalty associated with mechanical disks.

Even on systems where this overhead exists, most real world GA problems will have fitness functions so complex that they will take much longer to evaluate than the overhead. In such a situation, the file transfer mechanism will only play a small part in the time taken to solve the problem.

For problems with simple and fast objective functions, this overhead will constitute a substantial part of the problem run time. But simple problems such as these do not always need a GA to find a solution.

5.4. Implementation Details

The GA testbed system is implemented in three parts as shown in Figure 6. The code for the console window is contained in the Java class testbed.class. When the user has entered the GA parameters into this window, the "Run New GA" button is pressed which creates a GAPlotter object. This object contains the GA mechanism and creates the GA Results window. The code for this is contained in the GAResults.class file.

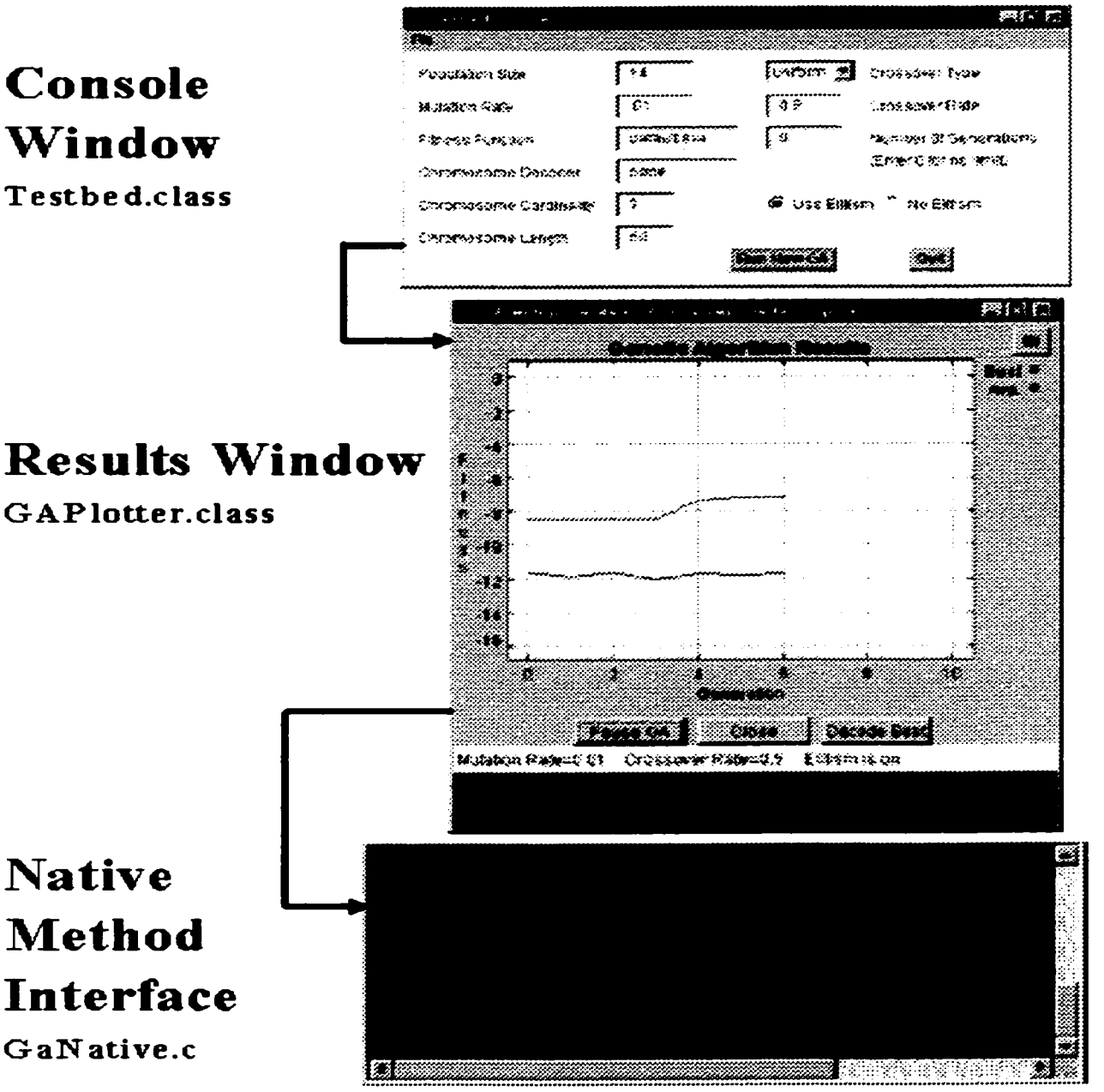

Figure 6 Three Part Implementation

Whenever the GA Results Window needs to access the objective function, a call is passed through the Native Method Interface. The code for this is in the file GaNative.c

These three main parts rely on help from other Java classes and third party library functions. The library functions are used to facilitate real time plotting and convenient graphical component placement. The complete source code for GA Testbed can be found in Appendix F.

5.4.1. The GA Testbed Console: testbed.class

The user interface is implemented using Java and is simply a window containing a system of menus and input boxes for collecting parameter information. It also launches an instance of the GAPlotter window whenever the "Run New GA" button is pressed. Information collected from the various menu boxes is passed to the GAPlotter when it is instantiated.

After creating a GAPlotter window, the console window does not close but stays open in case the user would like to create another GAPlotter window. There is no limit on the number of GAPlotter windows that can be open. Each window will be running an independent GA with independent

parameters. In this manner the GA testbed can be used for solving more than one problem at a time.

This may be a useful feature in certain circumstances, but running concurrent GAs on a single CPU will slow the solution time of each of the genetic algorithms. On a muti-processor platform though, this could offer a significant advantage.

The testbed console is a Java class called testbed.java and is implemented as a series of Java panel objects. The panels are located on the window surface by the position layout manager where the position is determined by a set of constraints. Each of the menu items requires two components, a *label* object to provide a text message, and in most cases a *text box* object. These elements are assigned a position according to a row and column identification where they are displayed in their appropriate place.

The majority of the GA parameters are entered through simple text boxes. When these boxes are created, a size and default parameter value must be specified. The size sets the number of character spaces available

for input and the default value specifies an initial value that may or may not be changed by the user.

There are a few GA parameters that are not input via a text box. The crossover type is selected using a drop down selection list created by using the Java *choice* object. The items in the drop down list that can be selected are created using *the add item* method from the *choice* class.

The buttons used to select whether elitism is to be used are created using the Java *check box group* object. Within this check box group there are defined two *check boxes*, one for true and the other for false.

In addition to the GA parameter selection items, there are two buttons on the console screen. The first is labeled "Quit" and is used to completely quit the console and all running GA instances. The other button is labeled "Run a New GA " and is used to start a genetic algorithm by launching a GAPlotter window with the desired GA parameters.

These buttons are created using the Java *button* object and placed on the console window using appropriate position constraints supplied to the position layout manager. Since these buttons are event items they are

attached to an action listener using the Java Development Kit version 1.1 event model.

An action event method is defined to handle and process these events when they occur. The event associated with the quit button simply terminates all objects and closes the console window, while the event associated with the "Run a New GA" button, retrieves the entered or default information from all of the parameter selection check boxes, choice menus and text boxes. These parameters are converted as required into the proper data type, and a new GAPlotter instance is created and these parameter are passed to the GAPlotter constructor method. The Gaplotter object counter is incremented so another independent instance will be created if the "Run a New GA" button is pressed again.

5.4.2. The GA Results Window: GAPlotter.class

The GA Results window is displayed when a GAPlotter object is instantiated. It contains the main code for running the GA and displaying the results on the window. Results are displayed as a real time plot of the best

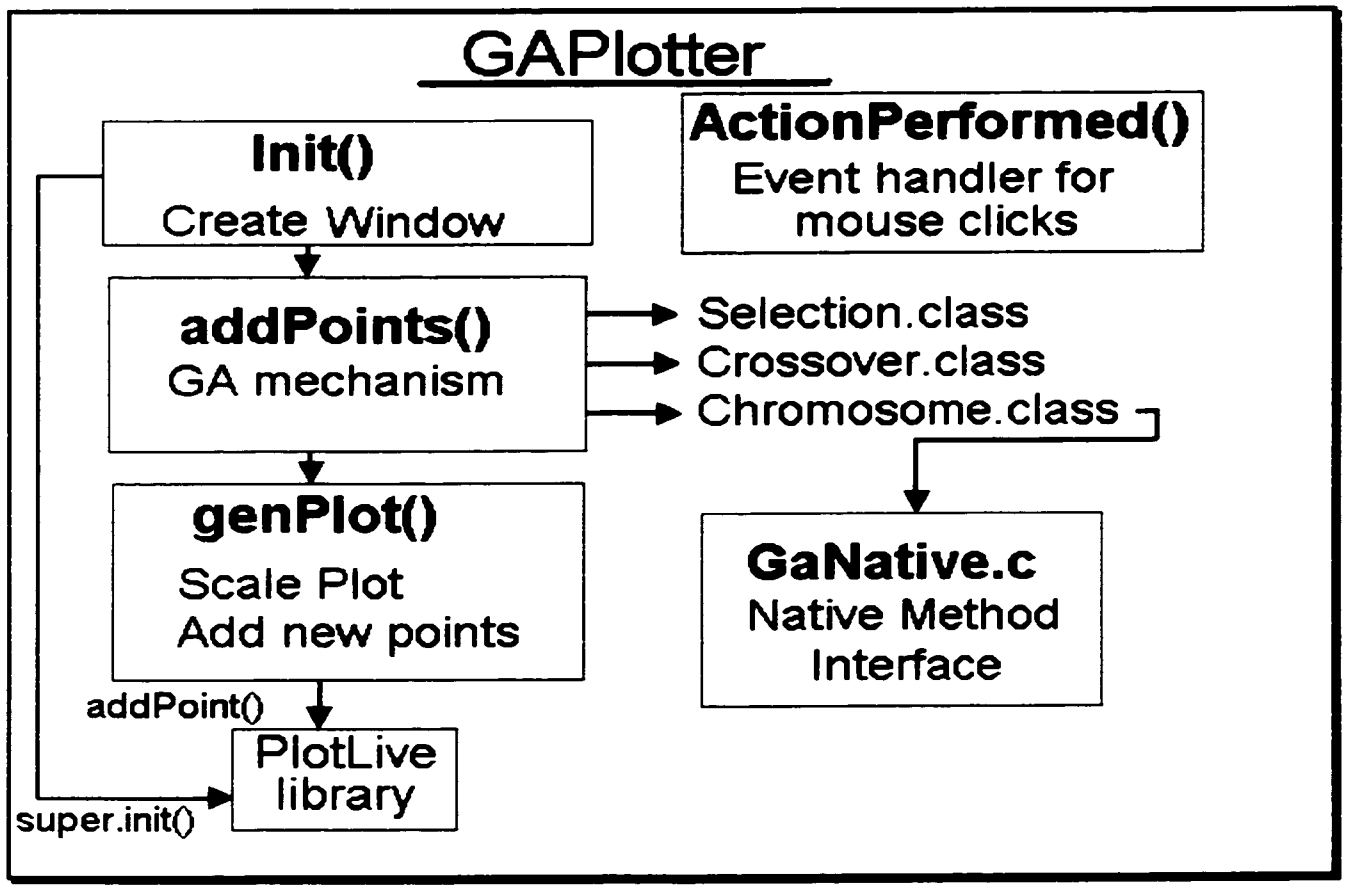

Figure 7 GAPlotter Structure

and average fitness values versus the generation number of a population and the best chromosome and it's fitness value are printed on the Window.

Several data structures are used in the form of Java classes. These include a Chromosome class for the storage of genes and related data items as well as the methods used to manipulate them. The structures pertaining to selection and crossover are also encapsulated, each in their own separate class along with their required methods.

The GAPlotter class inherits structures and methods from the PlotLive library class which are used for the real time plotting. A frame object is used as the base for the graphical objects with panel object acting as intermediary place holders to get the proper alignment. The graphical object placement is under the control of several *BorderLayout* layout managers.

Buttons are included to close, pause and decode the chromosome, each with its own action event listener. The event triggered by pressing the *Pause* button simply toggles the Boolean value in the pause variable. When the pause variable is true the main GA is placed in an infinite loop. Setting it to false will release the loop and the GA will continue. The label shown on the button is also toggled from *Pause GA* to *Resume GA*.

5.4.2.1. The Initialization Method: *Init()*

Once the starting variables and data structures have been initialized, the method init() is executed. This is used to set up the parameters for the graph such as the title and initial plot ranges. With these values set, the super.init() method is invoked to call the initialization method of the parent class. Since this is the plotLive library class, its init() will be executed, which will created the graph and start the data graph data acquisition phase.

Once the PlotLive library has initialized the empty graph with the title and axis, it will look for data values. The convention it uses for this is to call a non-library method called addPoints() which is expected to calculate all the X & Y points and pass them back to PlotLive, a pair at a time, with a call to PlotLive's addPoint() method. Note that these two have similar sounding names. The addPoints() method is part of the GA testbed code while the addPoint() method is part of the PlotLive graphing library.

*5.4.2.2.*The GA Cycle Method: *addPoints()*

The *addPoints()* method is where the data points are calculated for plotting so it contains all of the logic for implementing the actual genetic algorithm. It is called right after the PlotLive library function is initialized and the initial plotting window has been drawn.

Its first task is to create a new population of chromosomes by making an array of chromosome objects. The size of the array is set to match the population size and the length of the chromosome is set to the specified value. Since a chromosome object is responsible for evaluating it's own fitness, each new chromosome is passed the name of the command line fitness function during creation.

Once the GA is initialized, a GA-cycle loop is entered. This runs until the generation count exceeds the number of generations entered by the user or is an endless loop if the number of generations is specified to be 0.

At the top of the loop there exists a population upon which to create the next generation. It is either the initial population or the population created from the previous pass through the loop. This population is subjected to the selection mechanism. The selection mechanism is a completely self contained object within the Java environment and is explained in detail in a later section. The current population is passed to the selection mechanism which returns a new population exclusively consisting of selected individuals.

This population of selected individuals is then subjected to crossover. In a loop, each pair of chromosomes is considered in order. A

random number between 0 and 1 is chosen and compared with the user specified crossover rate. If the number is less than the crossover rate then the crossover operation is performed on this pair. Otherwise the pair is left untouched.

The crossover operation, like selection, is handled by a separate crossover object that has methods for each of the types which can be chosen from the crossover drop down list. These are described in detail in a later section. The two chromosomes in the pair under consideration are passed to the crossover operator which performs the operation and replaces the chromosomes with their offspring. The type of crossover desired by the user is passed to the chromosome object during its creation.

Once all the pairs of chromosomes within the population have been processed by the crossover operator, a check is performed to see if the user has specified that elitism is to be used. If so then a copy of the current best chromosome is used to replace the first chromosome in the population. There is no significance to replacing the first chromosome as any chromosome would do. Since the current best chromosome is from the previous or parental generation, this has the effect of carrying forward the

best individual from one generation to the next without destroying it through crossover.

With the population selected and crossed-over, mutation is now performed. For each individual a random number between 0 and 1 is chosen and if it is less than the user supplied mutation rate then the mutation operation is performed. This is handled by *mutateGene()* method in the population object and is described in detail in a later section.

This completes one pass through the genetic algorithm cycle and all that remains is a little bit of housekeeping. The population is searched for the best chromosome which is saved, the generation count is incremented and the *genPlot()* method is called to update the graph by adding the new points for this generation.

*5.4.2.3.*The Graph Control Method: *genPlot()*

The *GenPlot()* method is responsible for updating the graph with the results of the most recent pass through the GA cycle. The population average is calculated and this, along with the best individual in the population and the generation count, is passed to the *plotLive* library function.

This method is also responsible for scaling the range of the X and Y axes to make sure the graph does not run outside of the viewable area. This is done by checking the current range against the newly plotted values and if any axes are in need of correction then the corresponding range is doubled. In this way, the graph always occupies at least half of the available space in the direction of any axes.

This method is also responsible for updating the text at the bottom of the window showing the values of the current best chromosome and its fitness.

5.4.2.4.The Event Handler Method: *actionPerformed()*

Like other windowing languages, Java is event driven. When an event such as clicking a button occurs, an event handler is called to determine what action, if any, should be taken. The event triggered by pressing the *Close* button which simple destroys the button's *GAPlotter* parent window and all of that window's associated data structures using the *dispose()* method. The GA console and any other GAPlotter instances are left untouched.

The event triggered by pressing the *Decode Chromosome* button creates a new instance of the CallNativeGa class and passes to it the best

chromosome string in the current population. Since CallNativeGa was actually designed for calling the objective function, it requires a unique Identifier to be passed to it. In the case of a chromosome decoder this identifier is set to the word *decode* which signals the native method that this is not a normal objective function call but is instead a chromosome decoder call. The difference between the two types is that the objective function is passed a unique name to use as a file name while a chromosome decoder is not.

This chromosome decoder call executes the user supplied program while passing the chromosome string as the only argument on command line. This program should be designed to display the chromosome as a problem solution in some meaningful way. It must supply its own method of termination since once started, the testbed has no further interaction with it.

5.4.3. The Chromosome Object: chromosome.class

The Java class called chromosome defines the data structures used to hold a single chromosome string and all of its attributes as well as the methods needed to operate on it. The chromosome string is implemented as an array of Integers with each array element holding a single gene.

In order for the selection operator to perform a roulette wheel selection, the chromosomes must be ordered and their cumulative fitnesses must be calculated. For example, suppose we have a 4 chromosome population having fitnesses of 3 7 5 and 2 that are ordered as given. Their cumulative fitnesses would be 3, 3+7=10, 3+7+5=15 and 3+7+5+2=17. Each chromosome object stores this value along with the fitness value.

5.4.3.1. The Chromosome Initialization Method: *initializeChromosomeRandom()*

The method used to create a new chromosome with a random string is called *initializeChromosomeRandom()*. This simply assigns each position along the string with a randomly chosen value between 0 and the chromosome alphabet cardinality -1.

5.4.3.2. Determining a Chromosomes Fitness: *getFitness()* and *evalChromosome()*

The publicly available method *getFitness()* is used to determine the fitness of a chromosome and return its value. When it is called it checks to determine if this particular chromosome has been previously evaluated for fitness. If not, then a call is placed to the evalChromosome() method to evaluate it via a native method call to the command line objective function.

The resulting value is stored in case the *getFitness()* method is called again for this particular chromosome. In this manner, the objective function is only called once for each chromosome entity.

The method *evalChromosome()* is a protected method that is only called by the getFitness() method. It uses Java native methods which are discussed in detail later, to place a command line call to the objective function.

Before doing this though, the chromosome must be converted from an integer array to a string suitable for use as an argument to a command line function. Because the cardinality of the alphabet used for the string is user supplied, this conversion follows a scheme similar to hexadecimal. That is, integer values of 0 to 9 map to their corresponding numeric character; values 10 to 15 map to the letters A to F; and values of 16 and beyond map to letters beyond F, in the sequence 16=G, 17=H, etc. Using this conversion method places the upper limit of the chromosome alphabet cardinality at 36, which is far beyond anything needed in practice.

5.4.3.3. The Chromosome Mutation Method: mutateGene()

The mutation operation is actually performed within the chromosome class. This is best place for this since the chromosome class has all of the chromosome information necessary to perform this task.

Since the chromosome string can have gene values taken from an alphabet with any number of characters, the mutation operation involves more than flipping a bit, as is the case when binary chromosome strings are used exclusively. The technique used here is to randomly choose a gene to be mutated and the value of that gene is replaced with another value chosen randomly from the available members of the chromosome alphabet.

5.4.4. The Selection Object: Selection.class

The Java objects of the selection class are used to bundle together the data structures and the methods for performing fitness proportional selection. Also known as the roulette wheel method, the objective of this procedure is to select individuals randomly but with a biased probability. The probability of being selected is given by the expression (chromosome fitness)/(population total fitness).

To accomplish this, the total fitness of the population is calculated and then the relative fitness, defined as fitness/(population total fitness), of each chromosome is calculated. From this, the cumulative fitness can be calculated. This is dependent on the chromosomes order within the population. The absolute ordering is arbitrary and doesn't really matter as long as it does not change during the selection process.

A chromosomes relative fitness is calculated as the fitness of the individual plus the sum of the fitnesses of the ordered chromosomes that precede this one. Since cumulative fitness is based on relative fitness, none of the cumulative fitness values will be greater than one. These values are analogous to the pockets in a roulette wheel where the ball will stop, except that they are not evenly spaced since selection must be biased based on fitness.

With the chromosomes ranked, selection is accomplished by choosing a random number between 0 and 1. The chromosome with the cumulative fitness that is closest but larger than this number is the one chosen.

5.4.5. Crossover.class

The Java objects of the *crossover* class are used to bundle together the data structures and methods for the three types of crossover available.

Since the one point and two point methods are just special cases of the more general uniform crossover, the later is used exclusively. When either the one point or two point methods are selected, a special mask is created that will mimic these crossover types under uniform crossover.

The mask is implemented as a Boolean array so in the case of one point crossover, a point is chosen randomly and all those array elements up to that point are set to true and all others are set to false. In the two point case, the mask array elements between the two points are all set to true, while the others are all set to false.

Once the crossover mask has been established, it is simply a matter of dividing up the genes of each parent between the two offspring. The two offspring chromosome values replace the two parent chromosome values so that the parent population becomes the next generation.

5.4.6. The Command Line Interface: GaNative.c

The command line interface for calling objective and chromosome decoder functions is controlled by the C program GaNative.c. This is a very short program that performs some data type conversions and then issues a system() call to run the command-line program. Upon completion of this command, if an objective function was called it must get the returned fitness value. This value will be in the file with the unique ID given to the objective function so this file is opened, read and deleted.

This C language program is connected to the main Java program through the Java native method interface. To use this, a class is created called *CallNativeGa.class* containing a method called *runObjFunc()* which is declared to be of type native. With this designation, no code is needed for the method as this will be supplied by the C language program.

The naming convention is that a Java method defined as native will have a corresponding function in the C language program composed of three parts: the prefix *Java,* the method class name and the method name. Each of these is separated by an underscore. So in our case, the GaNative.c program contains the native method code in a function called *Java_CallNativeGa_runObjFunc.*

To use a native method the C language program GaNative.c is first compiled into a dynamic library called *ganative*. The Java program creates on object of the *CallNativeGa* class and issues a *System.loadLibrary("ganative")* command to load this library in the Java environment. From this point, calls can be make to the *CallNativeGa.runObjFunc*() method just as if it were an actual Java method.

5.4.7. Java Library Resources

In the construction of the GA testbed, several library packages are used. These include the Java libraries that come bundled with the Java Development Kit version 1.1.2 as well as two third party library packages: a real time plotting function and an absolute position layout manager.

5.4.7.1. The Real-time Plotting Function: PlotLive

The real time plotting is accomplished using the PlotLive class from a library package called Ptplot. Ptplot is a 2-D graphing component designed for use in applets and applications that was developed at UC Berkeley by Edward A. Lee, Christopher Hylands and William Wu. It is available for download at: http://ptolemy.eecs.berkeley.edu/java/ptplot.html.

In use, the GAPlotter class extends the PlotLive library class so that it inherits all of the plotting methods and structures. When PlotLive is initialized with the *super.init()* call from GAplotter it draws the empty plot frame. It then calls the addPoints() method, which is the main GA code in GAplotter. GAplotter performs the GA calculations and for each generation calls the plotLive *addPoint()* method to add the best and average data points to the plot. GAPlotter also keeps track of the plot scale and makes calls to plotLive methods as necessary to keep the whole plot visible and scaled properly.

5.4.7.2.The Absolute Position Layout Manager: PositionLayout.class

Placing buttons and other graphical objects within a display window using Java can be problematic. This is due to the lack of a Java position layout manager for setting fixed X and Y coordinates and fixed widths and heights. To get the proper spacing, a custom constraint-based layout manager for placing components within the window frame is used. It was written by Scoot M. Consolatti of IBM and is available for download at http://www.ibm.com/Java/education/position-layout-manager.html.

In use, a positionLayout object is created and attached to the current window with the *setLayout()* method. Then for each component to be placed

on the current window a new PositionConstraints objet is created and its X and Y values are set to the required coordinates for placing the component. To place the component, both the component object and the PositionContraints object are passed to the layout manager using it's *setConstraints()* method.

5.5. Conclusion

The GA testbed system was implemented using several Java classes, each with its own contribution to the overall system functionality. Each of the two user interface windows has its own Java class. There is a C program for the Java Native Method interface to user supplied functions. As well, two external third party library functions are used to facilitate real time plotting and convenience.

6. Testbed Verification And Results

6.1. Introduction

Assuring the accuracy and completeness of the GA testbed was challenging. A genetic algorithm is based on many random choices and events. This makes it very hard to test since there is no way to predict in advance what output is supposed to be produced. In normal deterministic programs, it is possible to predict which output values will be produced for a given input but with genetic algorithms, which use so many random operators, the output will be different for every run.

In order to overcome this problem it was necessary to test individual modules or units of the algorithm. By concentrating on a small section of the algorithm, it is possible to determine that that section is giving a correct answer even when random elements are involved. By doing this for all of the sections of the algorithm it is possible to gain some confidence in the accuracy of the system.

Verifying that the modules are all operating correctly does not guarantee that the whole system will operate correctly once it is assembled.

So, to test the accuracy of the whole system, a number of test problems are used to make sure the GA would find a solution.

The first two problems are chosen for their simplicity and the ease at which the results can be observed. They are a bit contrived but the GA does not care since all it knows about the problem is the fitness of each chromosome. The GA itself functions the same on simple or difficult problems with differences between the two only affecting the objective function.

The programming code for each of these test problems is given in Appendix G.

6.2. Example Problems

6.2.1. Maximum Ones Problem

The first problem that was chosen is called the maximum ones problem. The objective is to find the chromosome string with the most 1 digits. It is designed so that a person can easily evaluate the progress of the GA by observing the chromosome string. The fitness is obvious by simply looking at the number of 1's appearing in the string. It also has the

advantage of working with not only binary strings, but of strings with an arbitrary number of symbols in its chromosome alphabet. The objective function was also written so as to work on any length string allowing for the testing of the adjustable string length feature of the GA testbed.

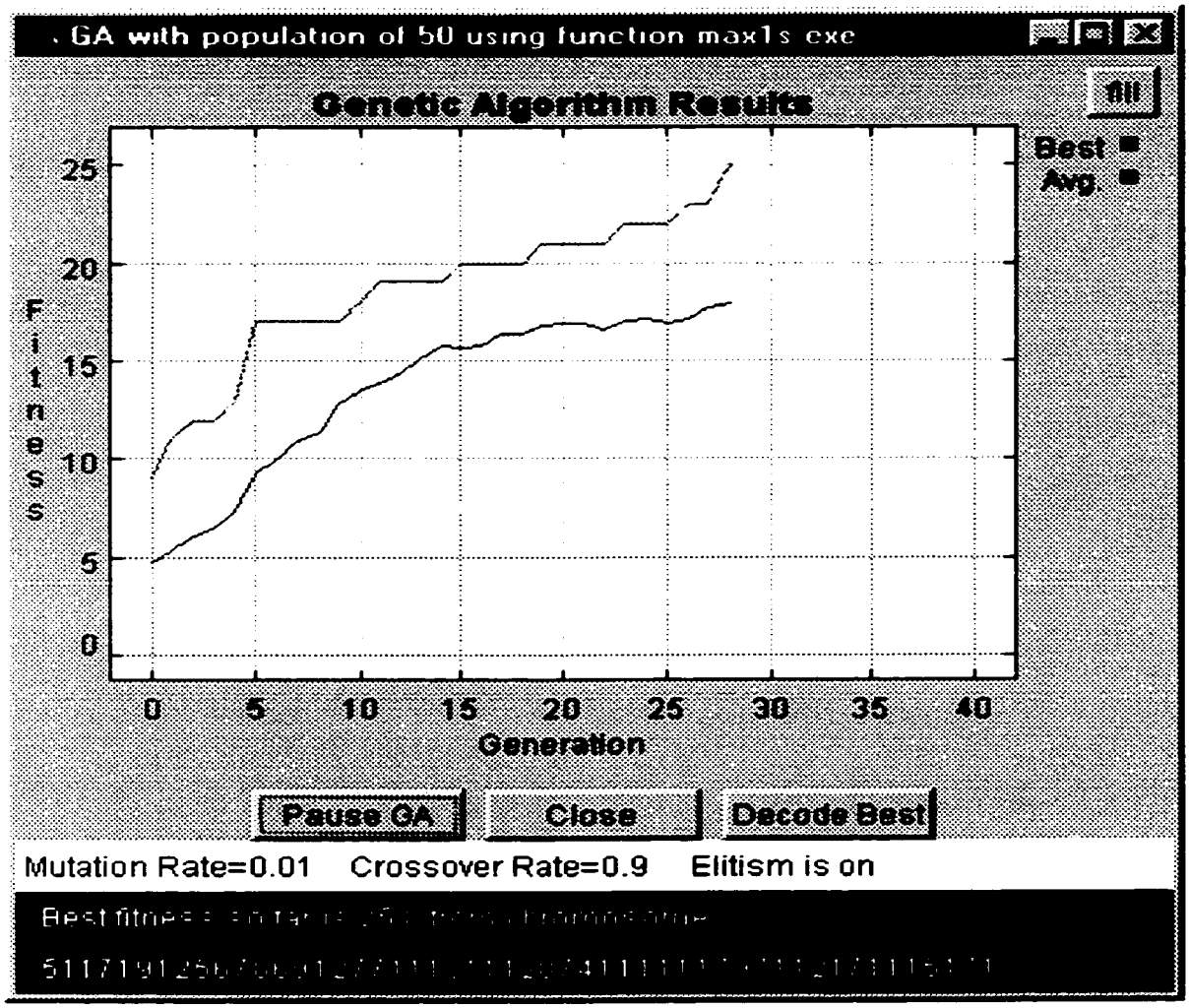

Figure 8 Sample Run of the Maximum Ones Problem

The output from a representative sample run is shown in Figure 8. In this run, the chromosome is a 50 character decimal number making for large search space. The GA has gone from an initial chromosome containing eight 1's to a chromosome containing twenty five 1's in just 27 generations of a population of 50. This works out to making 50*25 = 1250 samples in a search space of 10^{50} or about a mere 1.25×10^{-49} percent of the space.

A chromosome decoder function is really not necessary for such a simple program but one was created anyway for testing. It simply print the best chromosome and its fitness value to the screen. The output for the above run is shown in Figure 9.

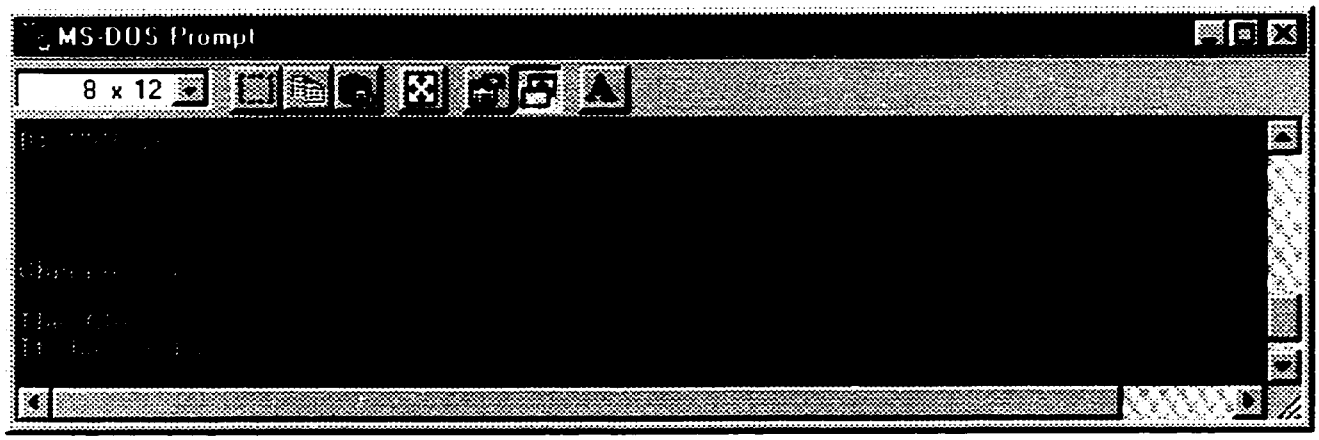

Figure 9 Chromosome Decoder Output for Max 1's Problem

6.2.2. Maximize X^{10}

Another problem commonly used in GA testing is to treat the chromosome as a number and raise it to some power. In this case, we have chosen 10 for the power and use a decimal number between 0 and 1. The solution is again very obvious with numerically larger numbers being better.

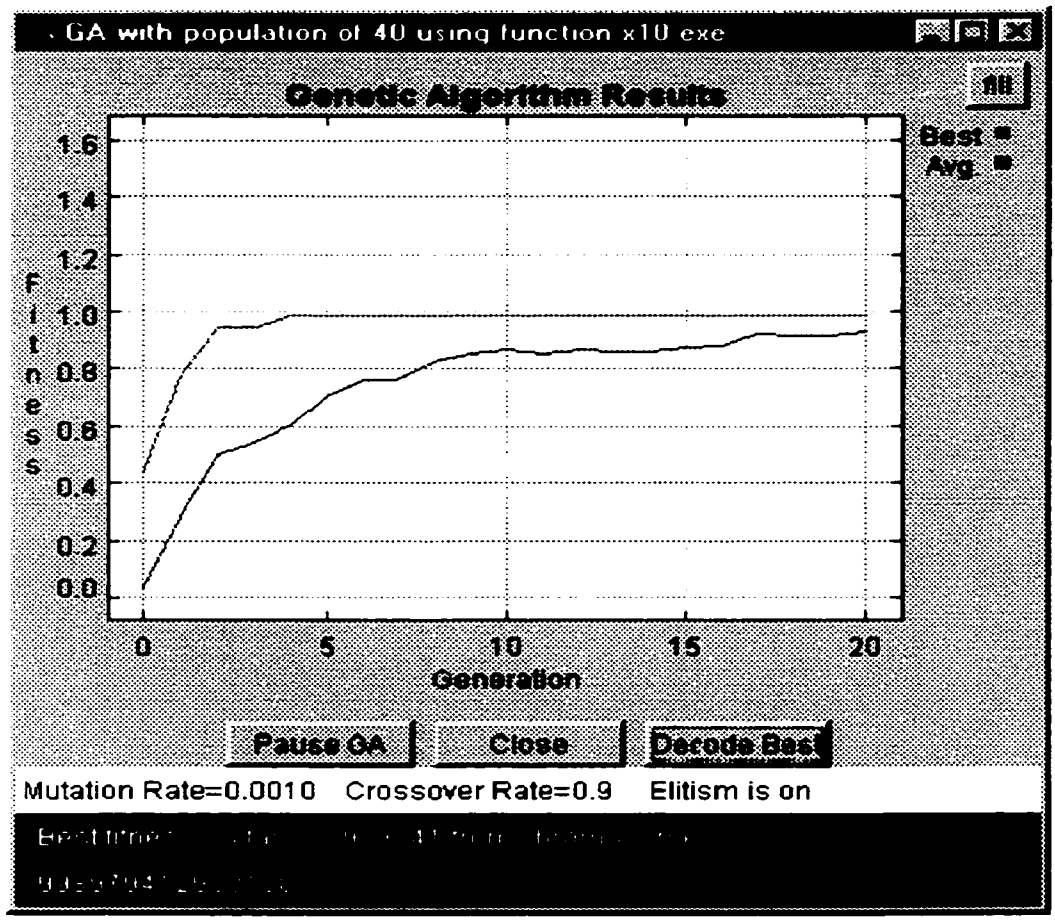

Figure 10 Sample Run of the X to the 10th Problem

A representative sample run is shown in Figure 10. In this run, the chromosome is a 15 character decimal number. This run demonstrates the GA characteristic that they can find good solutions fast but take a long time to find the best solution. Note that the GA has come to within 99% of the best value by the 4th generation, yet still hasn't found the best solution after 20 generations.

The decoder function for this simple example problem is not really needed. To make sure that the mechanism is working one was created anyway. It simply prints the chromosome string and its fitness value which is shown in Figure 11 for the above run.

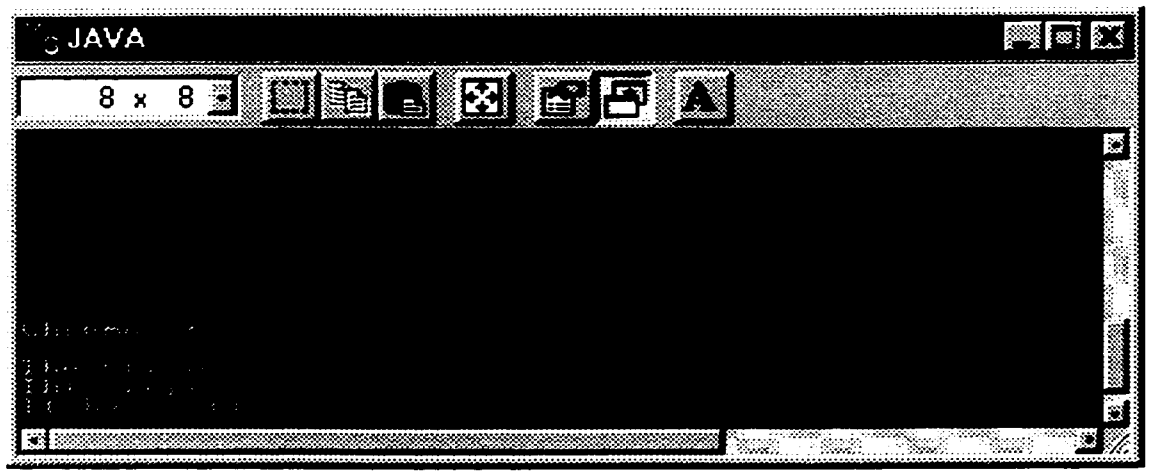

Figure 11 Chromosome Decoder Output for X to the 10th Problem

6.2.3. The Ten City Traveling Salesman Problem

The previous problems were chosen mainly as tools to easily evaluate the GA as it was being developed. The ten city traveling salesman problem, however, was chosen as an example of the type of problems usually encountered in practice.

The goal is for a hypothetical salesman to travel to ten cities while covering the shortest distance. The problem is constrained in that no city can be visited more than once. The chromosome coding method chosen is to use a 50 bit binary number and assign each city in the tour 5 bits to encode a "priority" rating. The cities will be visited in order of their priority ratings. A lethal chromosome is created if two cities have the same priority rating since this violates the problem constraint that the cities be visited in a sequence.

The fitness function calculates the distance traveled by a salesman. Since the GA testbed uses a larger number to indicate better fitness, we return the negative of the distance as the fitness. If a lethal is found its fitness is arbitrarily set to -9999 to indicate very poor fitness.

In order to monitor the GA's progress the cities were randomly selected to fall on a circle. The shortest path in such a situation will be

around the circumference of the circle. The amount of non-circular travel is easily determined by looking at a plot of the path. A representative sample run is shown in Figure 12.

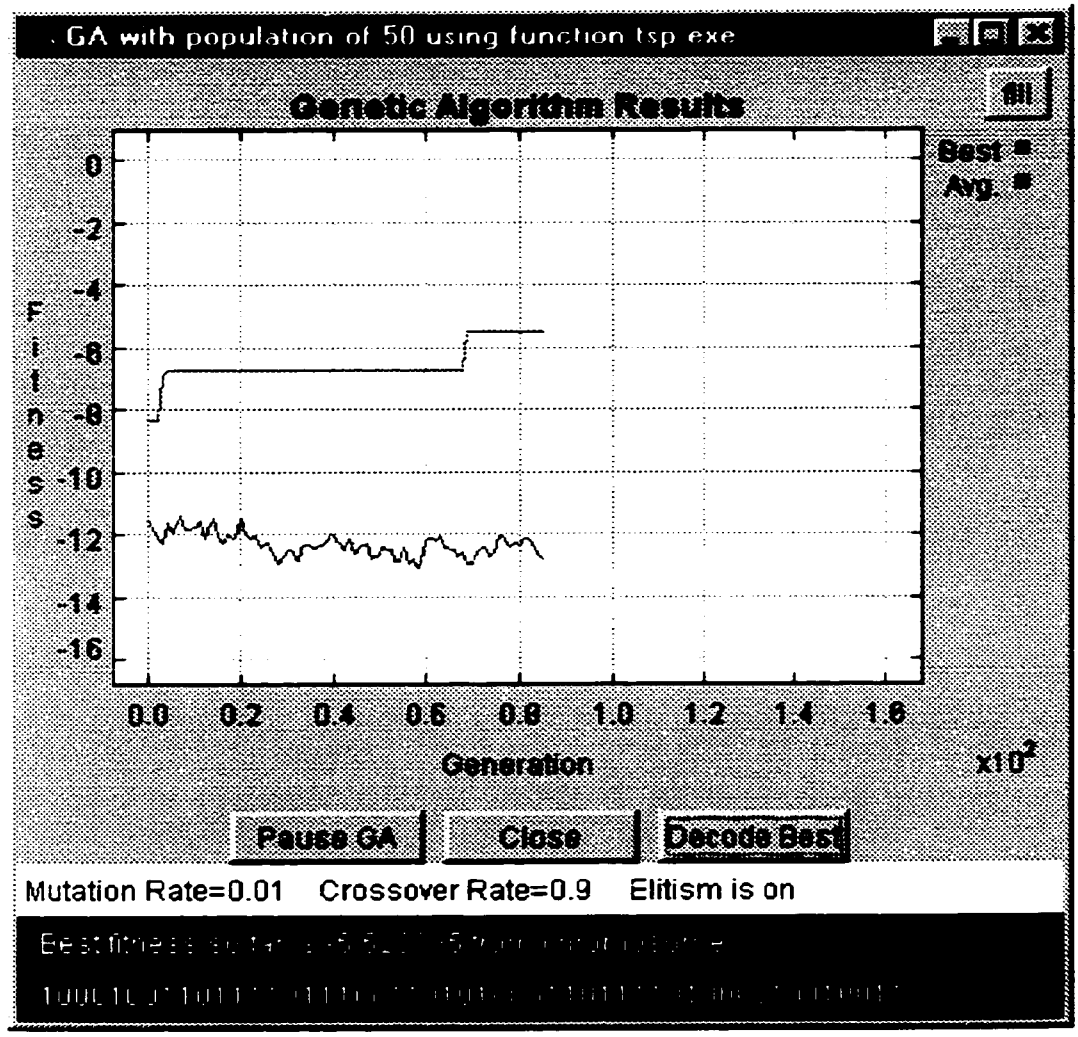

Figure 12 A Sample Run of the Ten City Traveling Salesman Problem

When solving this problem the GA testbed has found some very good solutions but it has never found the best solution even after runs of up to 12 hours.

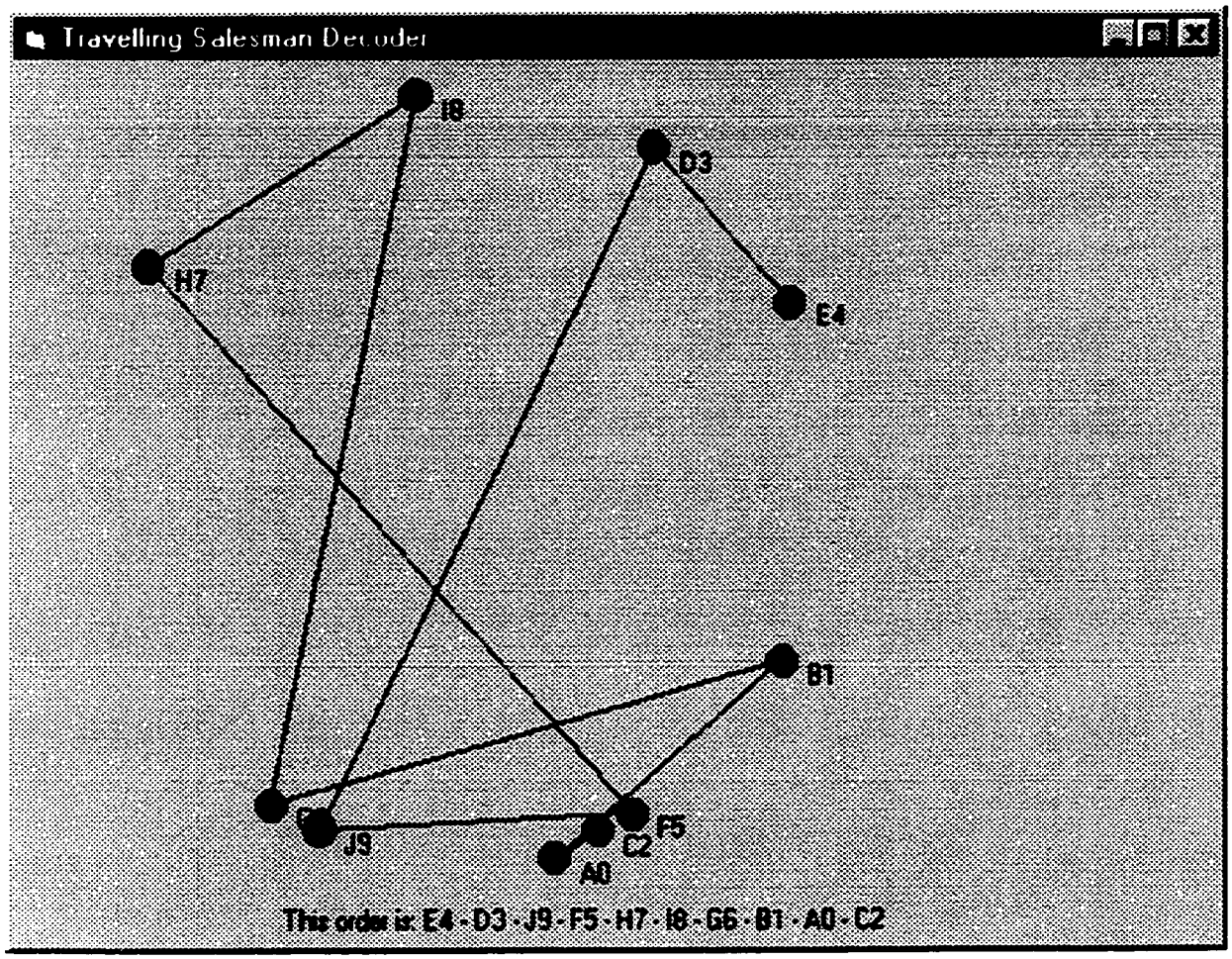

Figure 13 Poor Salesman's Path Found Early in the Run

The chromosome decoder for this problem was written in Visual Basic so a plot of the path could be made. Figure 13 shows a plot of the best

solution taken in the second generation before a good solution has been found. The cities are represented by red dots and have been labeled with the names A0, B1, C2, D3, E4, F5, G6, H7 and I8. The lines show the path traveled with the order also printed near the bottom. Note that the path is far from circular denoted a poor path.

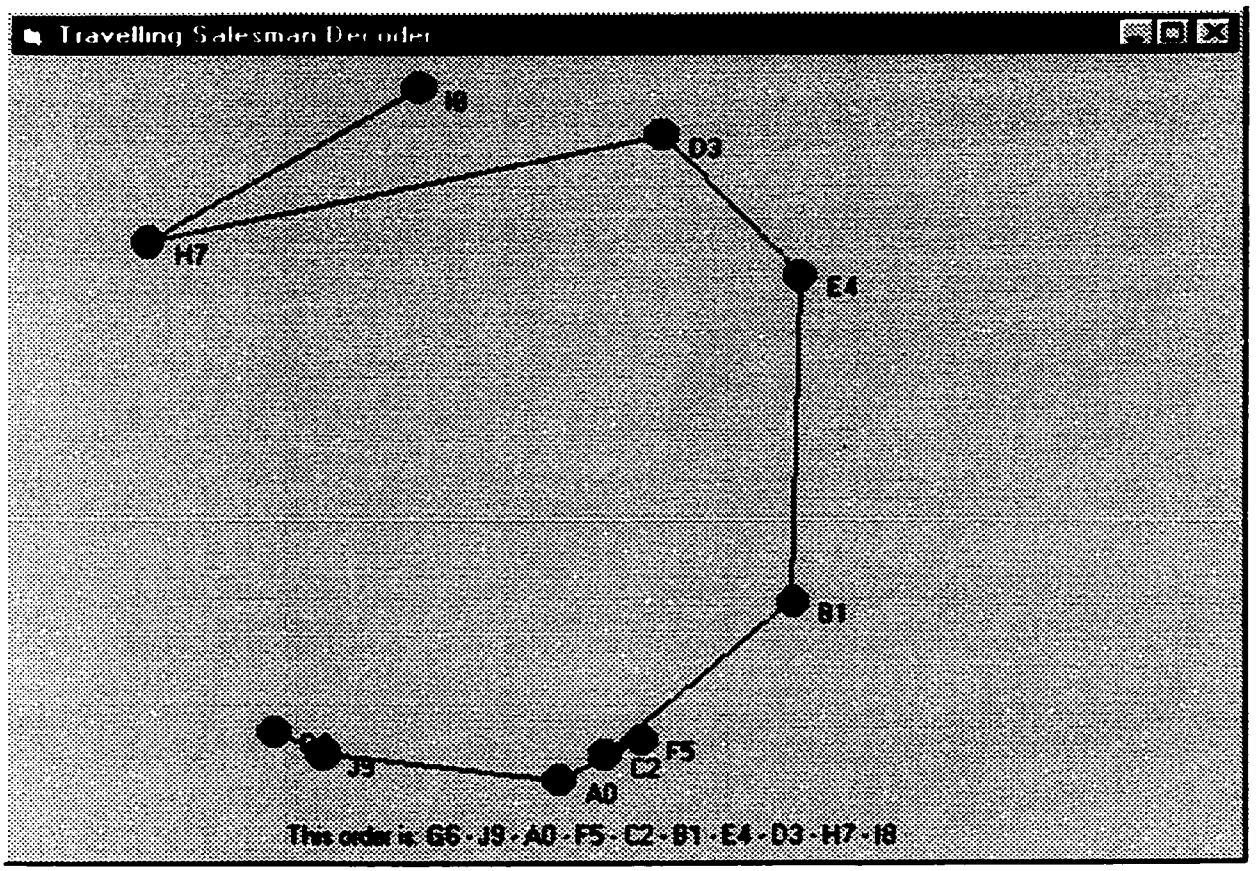

Figure 14 Good Salesman's Path Found After 90 Generations

Figure 14 shows a plot later in the same run as that of Figure 13. In this case the GA has run for 90 generations. Note that the path is more

circular indicating that it is getting close to the optimum which is a complete circle from H7 to G6.

6.3. Conclusion

Through the combined use of module testing and system testing, the accuracy and correctness of the GA testbed has been established. In all three of the system test problems, the GA over time, will consistently find better solutions. This provides the confidence that the necessary GA functioning has been achieved allowing us to conclude with near certainty that the testbed will perform properly with any GA suitable problem.

The GA testbed is obviously written in a platform independent language and interfaces to the user supplied functions at the command line level. The additional evidence that it functions properly is all that is required to establish that our goal of creating a simple, easy to use, GA development system has been realized. The GA testbed, therefore, meets all of the design criteria specified in the problem statement.

7. Conclusions and Future Work

The GA testbed implementation outlined here meets our objective of providing a complete genetic algorithm system which is capable of

performing all of the non-problem-specific calculations and has provisions for applying all of the commonly used algorithm variations. It is designed to run on any modern computer platform and to work with an objective function developed with virtually any programming language.

This enormous flexibility allows for the rapid prototyping of a problem solution. It far exceeds the capability of the traditional GA toolkit construction methods employed up to now. The combined flexibility and ease of use of this testbed should allow GA technology to be applied more widely, to more problems and by more people than ever before.

7.1. Testbed Limitations

As with any design exercise, the GA testbed described here has some compromises. These are mostly in the area of trading speed for flexibility. There are also certain capacity limitations but these are system dependent. For instance, there is a physical limit on the number of GA instances that can be simultaneously executing due to a physical limit on the amount of memory available. As well, the chromosome length will be limited by the size of the command line buffer. This is usually not a problem though since it will be in excess of 256 characters.

7.1.1. Java Limitations

In order to achieve the required platform independence it was necessary to develop the GA testbed using the only viable platform independent language: Java. Java is not a true complied language like C++. but instead its source code is compiled into an intermediary pseudo code. This intermediary pseudo code is then executed on target systems using a Java Virtual Machine (JVM).

The JVM on older systems is an interpreter that reads each piece of pseudo code and performs the appropriate action. This introduces a lot of overhead and tends to run programs very slowly.

More recent JVMs found on newer systems employ a method called *just in time* compiling which compiles the Java pseudo code into native machine code for execution at the full system speed. The task of compiling the pseudo code into real native code introduces some overhead of its own which tends to make Java programs run slower than equivalent non-Java programs.

These problems are not specific to the GA testbed but apply to all applications written in Java. There is a great deal of research currently

underway to address these problems. As the technology behind the JVM matures, it is expected that its speed will increase to the point where the difference between it and non-Java applications will be negligible.

7.1.2. Speed Limitations Due To Parameter Passing

In order to interface with an objective function that is actually a stand-alone program it is necessary to pass parameters at the command line level. For starting an executable program, parameters can be placed on the command line. Returning a parameter though is problematic. The only direct return mechanism available is the program exit code, which is an integer value.

This is not sufficient to handle an arbitrary fitness value as needed by the GA testbed. The alternative method, used here, is to have the objective function write the return value to a file so that after the return to the GA native method the file is opened and its contents read and the file then destroyed.

It is necessary to use a unique file name for every call so that multiple instances of a GA will not get files mixed up. To handle this, the GA creates a unique file name which is passed to the objective function as a command

line parameter along with the chromosome string. In this way, on each call to the objective function, the resulting fitness evaluation will get placed in a file with a unique name that is already known to the GA.

Using a file as a communication channel like this can result in a serious degradation of performance as each objective function call incurs a large overhead penalty. In a typical GA application though, the objective function is likely to be large and complex and take a while to perform. In such a situation, the overhead due to file creation and deletion will be a small percentage of the overall time spent on evaluating fitness. For simple problems, such as the maximum ones problem used for testing, this overhead does significantly degrade performance.

7.2. Future Work

There are several enhancements that can be made to the GA testbed. More GA options could be made available from the GA testbed console window. As well, increased flexibility could be achieved by allowing users to create their own functions to perform major GA cycle tasks such as selection, crossover and mutation.

The GA testbed speed performance could be improved. Improvements to speed should come automatically as Java and its JVM are improved, but the method communication, via a file, is slow and cumbersome and needs to be addressed. Perhaps some form of RAM disk can be used or perhaps extensions to Java will, in the future, allow output printed to the console by the objective function to be captured by the GA testbed. In any case, there is room for improvement in this area.

8. References

[Ack87] D.H. Ackley. An empirical study of bit vector function optimization. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 170--204. Pitman, 1987.

[Ang92] Peter J. Angeline. Antonisse's extension to schema notation. *GA-Digest*, 6(35):--, October 1992.

[Ant89] J. Antonisse. A new interpretation of schema notation that overturns the binary encoding constraint. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 86--91. Morgan Kaufmann, 1989.

[Ant92] Jim Antonisse. Antonisse's extension to schema notation. *GA-Digest*, 6(37):--, November 1992.

[Bac96] T.Back. *Evolutionary Algorithms in Theory and Practice.* Oxford university Press, New York, 1996.

[BBM93] D. Beasley, D.R. Bull, and R.R. Martin. A sequential niche technique for multimodal function optimization. *Evolutionary Computation*, 1(2):101--125, 1993.

[Boo85] L. Booker. Improving the performance of genetic algorithms in classifier systems. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 80--92. Lawrence Erlbaum Associates, 1985.

[Boo87] L. Booker. Improving search in genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 61-73. Pitman, 1987.

[Bra91] M.F. Bramlette. Initialization, mutation and selection methods in genetic algorithms for function optimization. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 100-107. Morgan Kaufmann, 1991.

[DC87] L. Davis and S. Coombs. Genetic algorithms and communication link speed design: theoretical considerations. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 252-256. Lawrence Erlbaum Associates, 1987.

[DS90] K. DeJong and W.M. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In H.-P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 38--47. Springer-Verlag, 1990.

[Dav85a] L. Davis. Applying adaptive algorithms to epistatic domains. *In 9th Int. Joint Conf. on AI*, pages 162-164, 1985.

[Dav85b] L. Davis. Job shop scheduling with genetic algorithms. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 136-140. Lawrence Erlbaum Associates, 1985.

[Dav90] Y. Davidor. Epistasis variance: Suitability of a representation to genetic algorithms. *Complex Systems*, 4:369--383, 1990.

[Dav91a] Y. Davidor. A genetic algorithm applied to robot trajectory generation. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 12, pages 144--165. Van Nostrand Reinhold, 1991.

[Dav91b] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[Dav91c] L. Davis. Bit climbing, representational bias and test suite design. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 18--23. Morgan Kaufmann, 1991.

[DeJ85] K. DeJong. Genetic algorithms: A 10 year perspective. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 169--177. Lawrence Erlbaum Associates, 1985.

[ECS89] L.J. Eshelman, R. Caruna, and J.D. Schaffer. Biases in the crossover landscape. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10--19. Morgan Kaufmann, 1989.

[ES91] Larry J. Eshelman and J. David Schaffer. GAs and very fast simulated re-annealing. *GA-Digest*, 5(37):--, December 1991.

[Esh91] Larry J. Eshelman. Bit-climbers and naive evolution. *GA-Digest*, 5(39):--, December 1991.

[Fog89] T.C. Fogarty. Varying the probability of mutation in the genetic algorithm. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 104-109. Morgan Kaufmann, 1989.

[GB90] D.E. Goldberg and C.L. Bridges. An analysis of a reordering operator on a GA-hard problem. *Biological Cybernetics*, 62:397-405, 1990.

[GD91] D.E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69--93. Morgan Kaufmann, 1991.

[GR87] D.E. Goldberg and J. Richardson. Genetic algorithms with sharing for multimodal function optimization. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 41--49. Lawrence Erlbaum Associates, 1987.

[Gol85] D.E. Goldberg. Alleles, loci, and the TSP. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 154--159. Lawrence Erlbaum Associates, 1985.

[Gol87] D.E. Goldberg. Simple genetic algorithms and the minimal, deceptive problem. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, chapter 6, pages 74--88. Pitman, 1987.

[Gol89a] D.E. Goldberg. *Genetic Algorithms in search, optimization and machine learning.* Addison-Wesley, 1989.

[Gol89b] D.E. Goldberg. Sizing populations for serial and parallel genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 70--79. Morgan Kaufmann, 1989.

[Gol90] D.E. Goldberg. The theory of virtual alphabets. In H.-P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 13--22. Springer-Verlag, 1990.

[Gre86] J.J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans SMC*, 16:122--128, 1986.

[Gre87] J.J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 42--60. Pitman, 1987.

[Gre93] John J. Grefenstette. Deception considered harmful. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms*, 2, pages 75--91. Morgan Kaufmann, 1993.

[Hol75] J.H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, 1975.

[Hol87] J.H. Holland. Genetic algorithms and classifier systems: foundations and future directions. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 82-89. Lawrence Erlbaum Associates, 1987.

[JM91] C.Z. Janikow and Z. Michalewicz. An experimental comparison of binary and floating point representations in genetic algorithms. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 31--36. Morgan Kaufmann, 1991.

[LR91] S.J. Louis and G.J.E. Rawlins. Designer genetic algorithms: Genetic algorithms in structure design.In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 53--60. Morgan Kaufmann, 1991.

[Lev91] J. Levenick. Inserting introns improves genetic algorithm success rate: taking a cue from biology. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 123--127. Morgan Kaufmann, 1991.

[MJ91] Z. Michalewicz and C.Z. Janikow. Handling constraints in genetic algorithms. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 151--157. Morgan Kaufmann, 1991.

[MS89] J. Maynard Smith. *Evolutionary Genetics*. Oxford University Press, 1989.

[Mic96] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs, Third Edition*. Springer-Verlag, 1996.

[Rud95] G. Rudolph. Convergence Analysis of Canonical Genetic Algorithms. Technical Report. University of Dortmund, Department of Computer Science, 1995

[SCLD89] J.D. Schaffer, R.A. Caruna, Eshelman L.J., and R. Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 51-60. Morgan Kaufmann, 1989.

[SD91] W.M. Spears and K. DeJong. An analysis of multi-point crossover. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 301--315. Morgan Kaufmann, 1991.

[SE91] J.D. Schaffer and L.J. Eshelman. On crossover as an evolutionarily viable strategy. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 61-68. Morgan Kaufmann, 1991.

[SG90] A.C. Schultz and J.J. Grefenstette. Improving tactical plans with genetic algorithms. *In Proc. IEEE Conf. Tools for AI*, pages 328--344. IEEE Society Press, 1990.

[SM87] J.D. Schaffer and A. Morishma. An adaptive crossover distribution mechanism for genetic algo- rithms. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 36-40. Lawrence Erlbaum Associates, 1987.

[SVG87] J.Y. Suh and D. Van Gucht. Incorporating heuristic information into genetic search. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 100-107. Lawrence Erlbaum Associates, 1987.

[Spe93] William M. Spears. Crossover or mutation? In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms*, 2, pages 221-237. Morgan Kaufmann, 1993.

[Sta87] I. Stadnyk. Schema recombination in a pattern recognition problem. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, pages 27-35. Lawrence Erlbaum Associates, 1987.

[Sys89] G. Syswerda. Uniform crossover in genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2-9. Morgan Kaufmann, 1989.

[Sys91] G. Syswerda. Schedule optimization using genetic algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, chapter 21, pages 332--349. Van Nostrand Reinhold, 1991.

[TMK96] K. S. Tang, K. F. Man, S. Kwong. Genetic Algorithms and their applications. *IEEE Signal Processing Magazine*, 13(6);22-37, November 1996.

[VL91] M. Vose and G. Liepins. Schema disruption. In R.K. Belew and L.B. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*. pages 237-242. Morgan Kaufmann, 1991.

[WSF89] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 133-140. Morgan Kaufmann, 1989.

Appendix A Abbreviations

GA Genetic Algorithm
JVC Java Virtual Machine
JDK Java Development Kit

Appendix B Definitions

Allele

Each gene occupies a specific character location within the chromosome string. Each gene position may take a character value called an allele.

Chromosome

A data structure consisting of a character string of coded task parameters.

Converged

A gene is said to have converged when 95% of the chromosomes in the population all contain the same allele for that gene. A population is said to have converged when all genes have converged.

It is commonly used to mean that the GA has slowed to a point that it doesn't seem to be finding new, better solutions.

Crossover

A reproduction operator which forms a new chromosome by combining parts of each of two parent chromosomes.

Deception

The condition where mating leads to reduced overall population fitness, rather than increased fitness. Proposed by Goldberg[gol89a] as a reason for the failure of gas on many tasks.

Elitism

A mechanism which is used to ensure that the chromosome of the most highly fit member of the population is passed on to the next generation without being altered by genetic operators. Using elitism ensures that the maximum fitness of the population can never reduce from one generation to the next. Elitism usually brings about a more rapid convergence of the population.

Epistasis

The interaction between different genes in a chromosome. It is the extent to which the contribution to fitness of one gene depends on the values of other genes.

Exploitation

The process of using information gathered from previously visited points in the search space to determine which places might be profitable to visit next.

Exploration

The process of visiting entirely new regions of a search.

Fitness

A value assigned to an individual which indicates how well the individual solves the task at hand. A fitness function is used to map a chromosome to a fitness value.

Gene

A position on a chromosome which usually holds the encoded value of a single parameter.

Generation

An iteration the creation of a new population by means of reproduction operators.

Genetic drift

Gene value changes resulting from chance rather than selection in a population over many generations.

Individual

A single member of a population which contains a chromosome representing a potential solution to the problem under consideration.

Mutation

A reproduction operator which forms a new chromosome by making random alterations to the values of genes. It usually occurs with low probability.

Offspring

An individual generated by any process of reproduction.

Optimization

The process of iteratively improving the quality of a solution to a problem as determined by a specified objective function.

Parent

An individual which takes part in reproduction to generate one or more other individuals, known as offspring.

Population

A group of individuals which interact together by mating to produce offspring.

Reproduction

The creation of a new individual from two parents.

Schema

A pattern of gene values in a chromosome, which may include `don't care' states.

Schema theorem

The fundamental theorem of genetic algorithms. It says that a GA gives exponentially increasing reproductive trials to above average schemata. The rate of schema processing in the population is very high, leading to a phenomenon known as implicit parallelism. This gives a GA with a population of size **N** an implicit processing factor of N^3.

Selection

The process by which some individuals in a population are chosen for reproduction. It is biased in favor of individuals with higher fitness.

Appendix C Genetic Algorithm Survey

C1. Abstract

This report examines the basic operation of Genetic Algorithms including the essential operations of selection, crossover and mutation. The theoretical foundations are reviewed including the fundamental theorem of Genetic Algorithms, the building block theorem and implicit parallelism. Research into advancements and extensions of the basic genetic algorithm is surveyed including refinements to the basic operators as well as techniques for managing GA difficulties such as deception, epistasis and genetic drift. This report also outlines the implementation of the simple GA as a Web based Java applet.

C2. Introduction

Genetic algorithms (GAs) are an attractive class of computational methods that are modeled on the mechanisms of natural evolutionary genetics. They seek to use the power of natural selection to solve difficult problems related to search and optimization.

The first rigorous study of the principles of GAs were reported by John Holland in his book *Adaptation in Natural and Artificial Systems* published in 1975 [HOL75]. This work has been subsequently extended by many others. They utilize methods that are similar to the methods found in natural selection to work on a population of problem solutions in an effort to find the fittest individual. It is hoped that this fittest individual is at or close to the optimal solution.

The technique is based on the observation that natural populations tend to evolve over many generations according to the principles of natural selection and survival of the fittest. Individuals in a population must compete with each other for a limited number of resources and ultimately for survival. The most successful individuals will be more likely to survive and thus to mate. The less successful individuals will be less likely to survive and thus will produce fewer offspring than a successful individual. This means that each succeeding generation will be more likely to inherit genes from successful individuals than from unsuccessful individuals.

Genetic algorithms borrow heavily from this natural evolutionary process to allow solutions to real world problems to evolve over many succeeding generations. Therefore, in order to artificially use the mechanisms of natural selection on an optimization and search problem, it is necessary to formulate the problem in line with that observed in nature. The solution to the problem must be expressed as a character string called a chromosome and there must also be a fitness function that can be applied to this string to determine the individual's fitness. For example, in the design of a bridge the chromosomes may represent the size and weight of certain beams and the fitness function would calculate the strength to weight ratio of a bridge built with these beams.

Within a population of individual solutions to a problem there are more fit and less fit solutions. Individuals are chosen from this population for mating depending upon their fitness score. Two chosen individuals are mated by cutting and splicing their chromosomes to form a new

chromosome. The offspring will thus inherit features from both parents. In this way the good characteristics of a population are transferred to each succeeding population while the bad characteristics are not. This results in the most promising areas of the search space being explored. If the problem has been coded into chromosomes properly, the population will converge to an optimal solution.

The GA is robust since the only requirement for applying it to a particular problem is that the solution can be expressed as a chromosome and there exists a fitness function to evaluate this chromosome's fitness. No other information about the problem is needed. This means that a GA can be applied to a wide variety of problems including some where there is no other solution technique.

A GA does not actually find a solution to a problem but instead creates new and better solutions based on existing solutions. Fortunately; the coding of a solution into a string as required by the GA allows initial solutions to be randomly generated.

While GAs are not guaranteed to find the global optimum, they are good at finding good solutions in a reasonable amount of time. However, when GAs are compared to specialized techniques for solving particular problems, they usually perform poorly in comparison. It is possible to combine a GA with a specialized solution technique where each searches the portion of the solution space to which it is most suited. The resulting hybrid is usually better than either approach alone.

C2.1 Classes of Search Techniques

Genetic algorithms are a type of optimization search technique. Search techniques in general, as illustrated in Figure 1, can be grouped into three broad classes [Gold89] calculus based, enumerative and random search. These can be further subdivided. Calculus based methods include direct and indirect. Indirect is the search for the peaks of a maxima by finding zero of the gradient. Direct techniques are those such as Newton's method. Random methods include simulated annealing, evolutionary strategies, genetic algorithms and the simple random walk through the search space. Enumerative methods are the brute force methods where all solutions in the whole search space are generated.

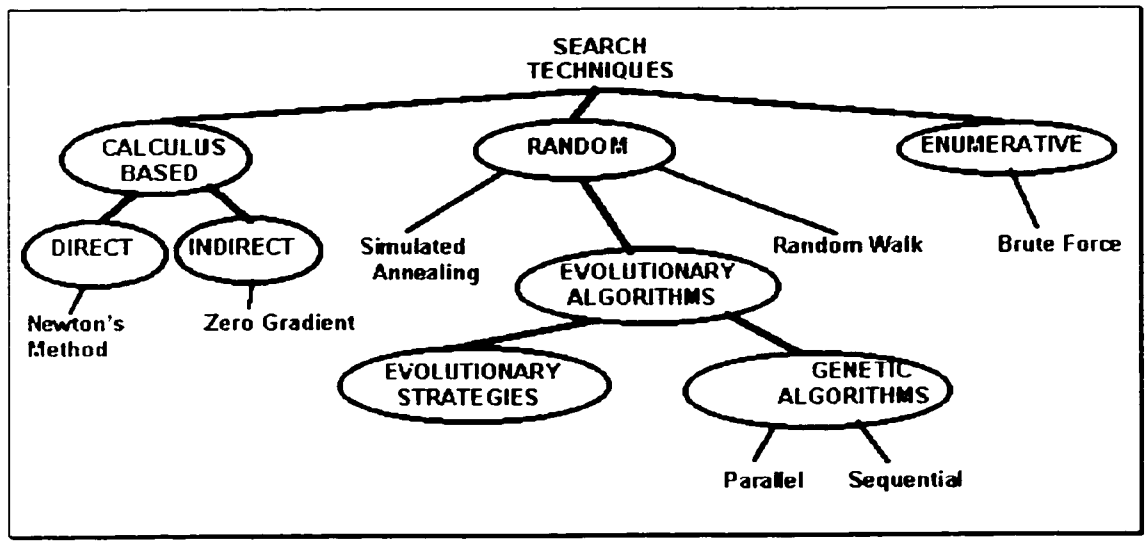

Figure 1 Search Techniques

C3. Simple Three Operator Genetic Algorithm

The basic genetic algorithm as described by Goldberg [Gol89a] is composed of three operators: reproduction, crossover, and mutation. These are applied to a population of individuals each of which is composed of a coded solution to the problem. The problem must have a fitness function that can be applied to each solution to determine its merit or fitness. Applying the operators to a population results in a new population with hopefully increased average fitness as well as an increase in the fitness of the fittest individual. This process is repeated until there is no further fitness increase at which time the solution is said to have converged. The pseudo code for the complete algorithm is shown in Figure 2.

```
BEGIN SIMPLE GENETIC ALGORITHM
Randomly generate initial population
Compute the fitness of each individual in the population
WHILE (NOT finished) DO
        // produce new generation
        FOR (population_size / 2) DO
                // reproduce 2 individuals
                Copy two parent individuals randomly selected from current generation
                        using probability biased to favor the fittest (Reproduction)
                Mate these copies by randomly splitting and recombining
                them to form two new offspring (Crossover)
                Remove the two original parents from current generation
                Place the two offspring into new generation
        END FOR
        Designate new generation to be current generation
        Randomly change some randomly selected individuals (Mutation)
        Compute the fitness of each individual
        IF (population has converged) THEN
                finished
        END IF
END WHILE
END GENETIC SIMPLE ALGORITHM
```

Figure 2 Simple Genetic Algorithm

C3.1 Problem Coding

The problem solution must be encoded in a form suitable for use with the reproduction, crossover, and mutation operators. The solution must be configured as a string of characters called a chromosome after its biological analogue. The characters may be taken from any fixed alphabet

that may be used to represent the problem. Holland [Hol75] has shown that a longer string is superior to a shorter string so a low cardinality alphabet is superior to a higher one. The lowest cardinality that will work is an alphabet of only two characters so a binary string is often used to code problems. In biology, chromosomes are made of strings of four different proteins so the biological alphabet has a cardinality of four.

C3.2 Fitness Function

The problem must also have an objective or fitness function that takes a solution chromosome and returns a value that represents a figure of merit for this solution. A higher figure of merit indicates a superior or fitter solution.

The genetic algorithm works on a population of chromosome strings each of which represent a solution to the problem. To begin an initial population of solution strings is chosen by some method such as simple random choice. This population is applied to the objective function and each solution is assigned a fitness value. The population is then subjected to the three operators of the genetic algorithm.

C3.2.1 Reproduction

Reproduction is the process of copying chromosome strings according to their fitness value. A chromosome string with a higher fitness value will have a higher probability of being copied. This is analogous to the natural selection process whereby an organism that is more fit has a higher chance of survival. To implement such an operator in algorithmic

form in a computer is possible in many ways. A common way is to create a biased roulette wheel where each chromosome string is assigned a slot on the wheel that is sized in proportion to its fitness. When the wheel is spun the chromosome string with the highest fitness will have the greatest possibility of being chosen.

The new population that results from the reproduction operator is biased towards those individuals that are the fittest. These selected individuals are now subjected to crossover. A crossover point is chosen at random as a positional number between 1 and the string length-1. Two new chromosome strings are now created by dividing the initial chromosome strings into two sections each at the crossover point and appending the first half of the first chromosome string to the second half of the second chromosome string and vice versa.

The mutation operator is now applied to the population. This operator applies a random alteration to the value of a chromosome string position. This alteration occurs with very small probability so that the likelihood of a bit actually mutating is very small. Mutation is necessary to replace genetic information that may have been lost or may not have actually existed in the original population to begin with. Its contribution to the outcome of the search is very minimal and is greatly overshadowed by the reproduction and crossover operators. In fact good results are obtained from genetic algorithms for probabilities of a bit mutating of one in a thousand.

C3.3 Example Problem

The simplicity and power of genetic algorithms can best be demonstrated with the step by step solution of a problem. The problem shown in Table 1 and

Table 2 is so simple it was solved by hand by Golberg [Gol89a] using nothing more than a coin to generate random numbers. Yet even with this simplicity it is fascinating to see the improvements possible in only a single generation. The problem goal is to maximize the function $f(x)=x^2$ where x is permitted to vary between 1 and 31 which is coded as a 5 bit binary number. To keep things manageable it is run with a population of only 4 individuals.

Table 1 Initial Population

String Number	Initial Population (Chromosome)	x Value (as Integer)	Objective Function Value $f(x)=x^2$	Probability of Selection	Expected Count	Actual Count (From Roulette Wheel)
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0 49	1 97	2
3	0 1 0 0 0	8	64	0 06	0 22	0
4	1 0 0 1 1	19	361	0 31	1 23	1
Sum			1170	1.00	4.00	4.0
Average			293	0 25	1 00	1 0
Maximum			576	0 49	1 97	2 0

Table 2 Mating

String Number	Mating Pool After Reproduction	Mate (Randomly Selected)	Crossover Site	New Population	x Value (Integer)	$\frac{F(x)}{x^2}$
1	0 1 1 0 | 1	2	4	0 1 1 0 0	0.58	144
2	1 1 0 0 | 0	1	4	1 1 0 0 1	1.97	625
3	1 1 | 0 0 0	4	2	1 1 0 1 1	0.22	729
4	1 0 | 0 1 1	3	2	1 0 0 0 0	1.23	256
Sum						1754
Average						439
Maximum						729

C4. Analysis of Simple Three Operator Genetic Algorithm

The previous section demonstrated the mechanics of the simple three operator genetic algorithm. By mimicking nature, researchers have developed a procedure that seems to provide useful results. However, to understand why it works requires a complete theoretical analyses.

This theoretical basis for the GA's operation was first worked out by Holland [Hol75] and later embellished by Goldberg [Gol89a]. It provides a thorough, generalized analysis of the operation of GA's. This analysis is often called the schema theorem, but its real importance is underscored by its other commonly used name: the fundamental theorem of genetic algorithms.

C4.1 Schemata

In order to analyze the workings of genetic algorithms it is necessary to have some method of describing a subset of a string. A subset can be described using the similarity template called a schema (in plural form they are called schemata). This is a string composed of the letters of the given chromosome alphabet plus a special symbol, usually *, that is used to indicate a don't care position in the chromosome string. A schema can be thought of as a pattern matching devise as it matches the particular chromosome string if in every location a 1 in the schema matches a 1 in the chromosome string and a 0 in the schema matches a 0 in the chromosome string and a * matches either.

For example consider the case of a binary string of length 5. The schema * 1 0 1* matches only the chromosome strings 01010, 01011, 11010 and 11011.

C4.2 The Fundamental Theorem Of Genetic Algorithms

The schemata theorem or the fundamental theorem of genetic algorithms is one of the most important properties related to genetic algorithms. In order to analyze the operation of genetic algorithms it is necessary to be able to count the schemata present within a population of strings and determine which grow and which decay during each generation. This is done by considering the affect of reproduction, crossover and mutation on a particular schema. The objective is to quantifying the GA's simultaneous manipulation of a very large number of schemata.

C4.2.1 Notation

In this analysis it is considered, without loss of generality, that strings are composed of characters from the binary alphabet V= {0,1}. For notational purposes strings will be referred to by capital letters and individual characters in the string by lower case letters. These individual characters may be subscripted by their position in the string as in $S=s_1s_2s_3s_4$. With this notation the a_i would be analogous to biological genes while the value of the a_i would be analogous to biological alleles. Populations of individual strings will be denoted as P_j, j = 1, 2,. . .,n . Each population existing at a time or generation t and will be denoted as **P**(t) where the boldface denotes a population rather than a string.

In order to describe the schema contained in individual strings and populations the three letter alphabet V+ = {0,1,*} will be used. The additional character * is used as a don't care or wild card symbol which will match either a 0 or a 1 at any particular string position.

For a string of length l there are 3^l schema which are defined over it. In general, for an alphabet of cardinality j there are $(j + 1)^l$ schemata.

The order of a schema is denoted by $O(H)$, and is the number of fixed (as opposed to wild card) positions that it has. For example, a schema of length 5 and order 3 is 1 1 1 * * .

A schema H will also have a defining length denoted by $\delta(H)$. This is the distance between the first and last fixed string position. For example the schema 1 * 1* * has a defining length $\delta(H) = 2$ because the first fixed

position is 1 and the last fixed position is 3 and 3-1 = 2. Similarly, the schema * 1 * * * * would have a defining length of $\delta(H) = 2 - 2 = 0$.

C4.2.2 Analysis

The previously discussed notation will be used to discuss the affect of reproduction on the expected number of schemata in the population. Suppose at a given time t there are m examples of a particular schema H contained within population **A**(t). This can be written as m = m(h, t). During reproduction a string A_i gets selected for copying with a probability of $p_i = f_i / \sum f_i$. Once the non-overlapping population of size n is chosen with replacement from the population **A**(t), there should be m(H, t + 1) representatives of the schema H in the population at time t + 1 as given by the equation:

$$m(H, t+1) = m(H, t) \bullet n \bullet f(H) / \sum f_i .$$

If $f(H)$ is the average fitness of the strings representing schema H at time t and since the average fitness of an entire population may be written as $\bar{f} = \sum f_i / n$ the reproductive schemata growth equation may be written as:

$$m(H, t+1) = m(H, t) \frac{f(H)}{\bar{f}} .$$

This shows that a particular schema grows as the ratio of the average fitness of the schemata to the average fitness of the population. Schemata with fitness values greater than the population average will be present in greater numbers in the next generation while schemata with

fitness values lower than the population average will be present in lesser numbers in the next generation. This operation is carried out with every schema in a particular population **A** in parallel. All schemata in the population grow or decay according to their schemata averages under the operation of reproduction. This simple operation of reproduction on the strings in a given population results in many more operations being performed on many more schemata.

Individual Schema will increase or decrease in number from generation to generation depending upon their fitness values. The exact rate of this growth or decay can be determined from the Schema's difference equation. Suppose that a particular schema remains above average by about $c\bar{f}$ with c a constant. The schema difference equation can then be rewritten as follows:

$$m(H,t+1) = m(H,t)\frac{(\bar{f}+c\bar{f})}{\bar{f}} = (1+c)\bullet m(H,t).$$

Starting at $t=0$, and assuming c is constant from generation to generation, the following is obtained:

$$m(H,t+1) = m(H,0)\bullet(1+c)^{t}.$$

This equation has the same form as the compound interest equation. It is a geometric progression which shows that reproduction allocates exponentially increasing or decreasing numbers of schemata. It can also be seen that for a schema that is above or below average, reproduction will allocate exponentially increasing our exponentially decreasing offspring in subsequent generations.

It is clear from the reproduction operation that subsequent generations will have exponentially increasing numbers of schemata that are fit and exponentially decreasing number of schemata that are not fit. This operation serves to concentrate the existing good solutions while eliminating some of the less good solutions. It does nothing to find new and possibly better solutions. This is the job of the crossover operator.

Crossover allows for a randomized exchange of information between strings. It creates new strings while preserving the allocation strategy pursued by reproduction. The result is an exponentially increasing or decreasing collections of particular schema throughout the population.

During the operation of crossover some schema are more likely to survive than others. For example, the schema * * * 1 0* * can only be destroyed if the crossover point is chosen so that it falls between the 1 and the zero at the 4th position. On the other hand the schema * 1 * * * * 0 can be destroyed if the crossover point falls between the 1 and the 0 at any of positions 2,3, 4, 5 or 6. This likelihood of survival is based on the defining length of the schema and is given by the crossover survival probability p_c.

The lower bound of the crossover survival probability p_c can be calculated under simple crossover as $p_c = 1 - \delta(H)/(l-1)$ since the schema is likely to be destroyed whenever a crossover site within the defining length is selected from the $l-1$ possible sites. Since the crossover is itself performed by random choice with a probability p_c at a particular mating, the survival probability may be given by the expression

$$p_s \geq 1 - p_c \bullet \frac{\delta(H)}{l-1}.$$

As can be seen, this expression reduces to the previous expressions whenever $p_c = 1$.

In order to calculate the number of a particular schema H expected in the next generation under the combined effect of reproduction and crossover, the following combined expression is used.

$$m(H, t+1) \geq m(H, t) \bullet \frac{f(H)}{\bar{f}} \left[1 - p_c \bullet \frac{\delta(H)}{l-1} \right]$$

The third and final operations performed by the simple genetic algorithm is mutation. This is the random alteration of a single string position with probability p_m. For a schema H to survive, all of its fixed positions must survive. Therefore, a single allele survives with the probability $1 - p_m$. Since each of the mutations is statistically independent, a particular schema survives when each of the $o(H)$ fixed positions within the schema survives. Multiplying the survival probability $1 - p_m$ by itself $o(H)$ times is the probability of surviving mutation as $(1 - p_m)^{o(H)}$. Since the probability p_m is very small $(p_m << 1)$ this can be approximated as $1 - o(H)(p_m)$.

Combining this with the previous expression gives the following relation describing the number of copies of a particular schema that can expect to survive into the next generation under all three operations of reproduction, crossover and recreation.

$$m(H,t+1) \geq m(H,t) \bullet \frac{f(H)}{\bar{f}}\left[1 - p_c \bullet \frac{\delta(H)}{l-1}\right] - o(H)p_m$$

This shows that short, low order, above average schemata receive exponentially increasing trials in subsequent generations. These schemata are often referred to as building blocks and their exponentially increasing importance to the outcome of a GA is often referred to as the building block theorem.

Golberg [Gol89a] uses this result to define a phenomenon called implicit parallelism that is used to explain the power of GAs. Here a GA, with a population of n, processes n chromosome for each generation. During these n calculations the GA is shown to be actually processing n^3 schemata. This exponential increase in the processing capacity is credited with giving GAs their superior abilities.

C5. **Advanced Techniques**

The simple genetic algorithm, as discussed so far, is the basic starting point for all genetic algorithms. Much research has gone into extending and improving the algorithm. This section outlines these advancements and refinements.

C5.1 Crossover Techniques

The simple GA performs crossover by making a single cut at the same location in each of the two parent chromosomes. This cut occurs somewhere between the first gene and the last gene. The cut sections are then exchanged to form two offspring. This method is somewhat simplistic and tends to destroy building blocks that contain widely spaced genes.

For this reason researchers have devised many new crossover techniques often using more than one cut point. The effectiveness of multiple-point crossover was investigated by DeJong [DeJ75] and he found that while 2-point crossover gives an improvement, adding further crossover points reduces the performance of the GA. As additional crossover points are added, the search becomes more random because building blocks are more likely to be disrupted. The problem space is searched more thoroughly at the expense of greatly increased search time. In the extreme, the search simply becomes a random search.

C5.1.1 2-Point Crossover

In 2-point crossover, chromosomes are arranged in loops by joining their ends together. Two cuts are made in the loop and the resulting segments are exchanged.

From this it it is clear that 1-point crossover is just a special case of the more general 2-point crossover where one of the cut points is fixed as falling between the last and first position. This would account for the increased performance seen when 2-point crossover is used. It is no more disruptive than 1-point crossover since they both have 2 cut points and 2-point crossover does not always destroy building blocks with widely spaced genes as is the case for 1-point crossover. That is, a chromosome treated as a loop with no beginning and no end can contain more building blocks since they are able to wrap around at the end of the string.

It is generally considered that 2-point crossover is superior to 1-point crossover.

C5.1.2 Uniform Crossover

Another form of crossover is the n-point uniform crossover where the number of points n varies dynamically with each mating. In this method, a randomly generated crossover mask is used to determine which genes of an offspring come from which parent. Each gene in the first offspring is created by copying the corresponding gene from one or the other parent according to the crossover mask. Where there is a 1 in the mask, the gene is copied from the first parent, and where there is a 0 in the mask, the gene is copied from the second parent. The process is repeated with the parents exchanged to produce the second offspring. A new crossover mask is randomly generated for each pair of parents. Offspring, therefore, contain a mixture of genes from each parent. The number of effective crossing points, while not fixed, will average L/2 (where L is the length of the chromosome).

For example, suppose we let the first parent be an arbitrary 10-bit binary string represented by the sequence **ABCDEFG** where **A** represents the most significant bit and **G** the least significant bit and similarly we let the second parent be represented by **abcdefg** then they would mate using uniform crossover as follows:

- A random crossover mask is chosen, (e.g. 0011101001)
- Wherever the mask has a 1 choose the corresponding bit from the first parent
- Wherever the mask has a 0 choose the corresponding bit from the second parent

- The resulting offspring is combined like this:

Parent	**A B C D E F G**	**a b c d e f**
Mask	**0 0 1 1 1 0 1**	**0 0 1 1 1 0**
Result	**- C D E - G -**	**a b - - - f**
Combined Offspring		**a b C D E f G**

•Reverse parents and repeat steps 2, 3, & 4 yielding the second offspring **A B c d e F g**

C5.1.3 Crossover Comparisons

There is much debate among researchers over which is the best crossover method to use. Syswerda [Sys89] favors uniform crossover since under uniform crossover, long defining length schema are less likely to be disrupted than under 2-point crossover. While short defining length schemata are more likely to be disrupted, Syswerda shows that the overall amount of schemata disruption is lower, thereby better preserving precious building blocks.

Under 2-point crossover the defining length, and not the order, of the schemata determines the likelihood of its disruption. While under uniform crossover, the likelihood of disruption of a given schema is based only on it's order and not its defining length. This means that the ordering of genes within a chromosome is completely irrelevant and eliminates the need for many of the re-ordering operators such as inversion (see section 5.4 Inversion and Reordering on page 24). Also, since the positioning

genes is immaterial there is no need to worry about coding the chromosome in such a fashion so as to create good building blocks.

An extensive comparison of 1-point, 2-point, multi-point and uniform crossover operators was performed by Eshelman et al [ECS89]. Theoretical analysis was performed in terms of positional and distributional bias on several problems. While they found that an 8-point crossover was good on the problems they tried there was only about a 20% difference in speed between the slowest and fastest techniques. From these results the choice of a crossover. operator would seem to be unimportant.

Theoretical analyses by Spears & DeJong [SD91] however, shows that 1- and 2-point crossover are optimal. They state that due to reduced productivity, 2-point crossover will perform poorly when the population has largely converged. Productivity is the ability of a crossover operator to produce new chromosomes that are different, thereby sampling new points in the search space.

If two similar chromosomes undergo 2-point crossover then the exchanged segments are likely to be identical causing the offspring to be identical to their parents. Under uniform crossover, this is less likely to happen.

They also describe a new 2-point crossover operator where the offspring are checked after crossover and if they are found to be identical to their parents then crossover is repeated using two new crossover points. When tested, this new operator was found to perform slightly better than uniform crossover. They later state [DS90] that this new 2-

point crossover is best only when there is a large population, and that for small populations uniform crossover is best due to the increased disruption that it causes.

C5.1.4 Other Crossover Techniques

Researchers have proposed many other techniques in attempting to increase the effectiveness of crossover. Following a lead from nature, several methods have been described [SM87],[Hol7],[Dav91a],[Lev91],[LR91] that varies the probability of crossover occurring at a particular string position. The crossover probabilities themselves become part of the chromosome so that the GA dynamically adjusts the sites that should be favored for crossover. With the crossover probabilities included as part of the chromosome, they are crossed over and passed on to descendants allowing the GA to learn which building blocks are more important than other genes in the chromosome.

There is another crossover operator designed for use in order-based problems called partially matched crossover (PMX). It was first described by Goldberg [Gol85],[Gol89a]. Order-based problems, such as the traveling salesperson problem, have fixed gene values with the fitness being determined by the order in which the genes appear. So in PMX the gene order is crossed instead of the gene values thus passing ordering information from parent to offspring eliminating the creation of offspring which violate problem constraints. Other work on order based operators was done by Syswerda [Sys91] and Davis [Dav91b] and custom tailoring

the crossover operator to make use of problem specific knowledge is discussed later in Section 5.11, Exploitation of Domain Knowledge.

C5.2 Mutation

Normally, mutation occurs with low probability and functions as a background operator [Boo87][DeJ85]. It is included to allow for the searching space that may otherwise be precluded by the converging chromosomes as genetic information is discarded during crossover. The exact amount of mutation necessary is somewhat open to debate. Too little and useful alleles that are not currently in the population can never be found while too much causes the GA to degenerate into a random search.

Although crossover is generally seen as the major mechanism for exploring new search space, there are examples in nature where creatures using asexual reproduction have evolved. When Schaffer et al [SCLD89] did a study to determine the optimum parameters for GAs, it was found that mutation played a larger role than previously thought.

In a study by Spears [Spe93], crossover and mutation were compared and it was found that each operator contained characteristics not found in the other but that each is simply a form of a more general exploration operator that modifies alleles based on available information. As the population converges, Davis [Dav91b] found that mutation plays an increasingly important role while the role of crossover diminishes.

Although its probability of use is small and it is seen as nothing more than a background operator, mutation plays a very important part in

a GA solution. Adjusting for the optimum GA parameters is difficult since changes in mutation rate will affect performance much more than changes to the crossover parameters [SCLD89].

C5.3 Naive Evolution

When crossover is eliminated completely from a GA it is said to perform naive evolution. Such an algorithm, using only selection and mutation, performs a hill-climbing type of search. Expanding on their previous work, Schaffer et al [SE91] studied this type of algorithm and compared it to some that use only crossover and no mutation. The results show that a crossover-only algorithm gives a much faster evolution than a mutation-only algorithm but a mutation-only algorithm finds better solutions than a crossover-only algorithm. Other researchers have reported successful application of naive evolution [EOR91],[ES91],[Esh91].

C5.4 Inversion and Reordering

For a GA to work effectively, the building block hypothesis requires that the genes be arranged in a particular order in the chromosome. To accomplish this, techniques for reordering the positions of genes have been developed. One of these techniques is inversion [Hol75] which reverses the order of genes between two randomly chosen positions within the chromosome. To keep track of a genes position within the chromosome some auxiliary positioning information must be

maintained with each chromosome. Gene reordering is an attempt to create chromosome codings with better evolutionary potential [Gol89a].

While the use of inversion is popular there has not been a lot of work done to justify it, or quantify its contribution. Its advantages have been shown when it is used on a very small task [GB90] but this analysis can not be extended to large tasks.

The inversion operator was copied from nature and belongs to a group of natural mechanisms known as karyotypic evolution [MS89]. These mechanisms along with others such as diploidy and dominance (see Section 5.10) are useful in nature where the competitive landscape is constantly changing and hence the objective function changes with time. In normal search and optimization problems with static objective functions, the advantages offered by these methods are questionable.

When uniform crossover is used, the ordering of genes is irrelevant so reordering would have absolutely no effect. When reordering is used, the search space is greatly expanded since the GA is searching for a solution to the original problem and simultaneously searching for the optimum gene ordering. The extra search time spent on ordering might be better spent on the original problem.

There may be applications where searching for the right gene ordering is the primary goal of a search. This may be desirable for setting up a GA coding scheme for a whole class of problems. In this case karyotypic evolution could be used. Another possibility is the meta-GA method used by Grefenstette [Gre86]. A meta-GA is a GA that works on a population where each individual chromosome represents a GA coding

scheme. The fitness function for the meta-GA derives it's fitness values by running the GA represented by each chromosome. Such calculation require a great deal of time to run and can only be justified when the results can be applied to many problems.

C5.5 Deception

The building block principle states that over succeeding generations there will be an increase in the number of chromosomes containing schemata that are also found in the global optimum until eventually these schemata will crossover into a single individual and the global optimum will be found. But on certain GA deceptive problems, this does not occur and schemata that are not in the global optimum increase in numbers faster than those that are.

This phenomenon has been studied in depth by many [Gol87] [Gol89a][Gre93][DG91] and it has been shown that the number of chromosomes containing a particular schema will increase if the schema's fitness is higher than the average fitness of all schemata in the population. The difficulty arises if the average fitness of schemata which are not contained in the global optimum is greater than the average fitness of those which are. This class of problem is deemed to be deceptive. A GA will usually, but not always, have difficulty solving a deceptive problem.

C5.6 Epistasis

A given gene's contribution to the overall fitness of an individual may be conditional on the value of other genes in the chromosome. Such

a gene would be called epistatic. In general, the amount of co-dependency among genes is termed epistasis and occurs in nature on a regular basis. For example, bats have a gene that gives them their keen hearing and another to make high pitched chirps. Either of these genes alone would not increase a bat's fitness but together they form a sonar system and have a major impact on fitness.

The amount of epistasis which occurs can vary from none to severe. An example of a problem with no epistasis is the counting ones task where the task is to maximize the number of 1s in the binary string. In this case each gene (bit) either has a value of 1 and contributes to the fitness or has a value of 0 and doesn't. An example of moderate epistasis is the plateau function where the fitness is 1 if all the bits in a chromosome are set to 1, and zero otherwise. Here the genes do interact but only for the global optimum when they all must be 1. Severe epistasis is where the genes interact in numerous and complex ways. An example of this would be a scheduling problem where the availability of a resource is dependent on other schedules.

Problems which exhibit no epistasis or even mild epistasis are easily solved by techniques like hillclimbing and do not require a GA [Dav91c]. When the problem has moderate to severe epistasis though, observations indicate that the other, simpler techniques do not perform as well as GAs. This is in direct contradiction to the building block hypothesis which states that for a GA to be successful there must be low epistasis. Davis & Coombs [DC87] point out that GAs have been made to work even in domains of high epistasis while Davidor [Dav90] states that present-day

GAs are only suitable for problems of medium epistasis since they lose effectiveness when applied to problems with high epistasis and simpler techniques, such as hillclimbing work better for problems with low epistsis.

It is possible to lower the epistasis of a given problem by changing the chromosome coding. Vose & Liepins [VL91] have shown that it is possible (although not necessarily easy) to code any problem in such a way that it has little or no epistasis. The effort in doing this though, might be more than that of solving the problem. Another method to lower epistasis is to use crossover and mutation operators that are custom designed for the individual problem. Here again, the effort may not be worth the results.

This type of problem recoding is demonstrated by Davis [Dav85a] where he converts a bin-packing problem, which finds the optimum positions for packing rectangles into a space, into an order problem, which finds the order of packing the rectangles instead. His algorithm uses intelligent decoding to apply domain knowledge to find good positions for each rectangle in the order specified by the chromosome. The chromosome of the converted problem has less epistasis than the original problem.

C5.7 Chromosome Alphabets

The fundamental theorem of Genetic Algorithms suggests that the strength of a genetic algorithm lies in the implicit parallelism of the operation. The algorithm works on many schemata at the same time.

Since a binary alphabet has the largest number of schemata of any alphabet, Goldberg [Gol89a][Gol89b] believes that it is the best. A different interpretation of schemata by Antonisse [Ant89] allows him to conclude that the opposite is true. He believes that high-cardinality alphabets contain more schemata than binary alphabets and so offer better performance. This conflict has caused much discussion [Ang92, Ant92].

To try to reconcile this conflict, Goldberg [Gol90] developed the theory of virtual alphabets to explain the good performance obtained with high-cardinality representations. This theory says that within the first few generations each symbol converges. These converged symbols can only take on a small number of possible values effectively making them a low cardinality alphabet.

Others have performed empirical studies of high-cardinality alphabets [Bra91,JM91,MJ91]. The major advantage seems to be that numeric problems can be coded directly, thus making it easier to develop problem specific crossover and mutation operators. The more specific information about a problem that can be incorporated in the solution mechanism, the better the solution will be.

In another study by Janikow [JM91], a direct comparison between binary and floating-point representations was done on numeric problems. Here the use of floating-point numbers resulted in faster and more accurate results. Since this finding was based on the solution of numeric problems the results can't be extended to problems where the parameters

are not numeric. Here there may not be any advantage to using high-cardinality alphabets.

C5.8 Dynamic Operator Parameters

The exact values of operator parameters, such as mutation rate, population size etc., that lead to the greatest GA performance is hard to pin down. As noted in Section 5.2, mutation has a greater effect toward the end of a search when the population has mostly converged. This suggests that the optimal value for each operator will vary during the course of a run and should be adjusted dynamically.

The use of decreasing crossover and increasing mutation has been reported to produce good results [Dav85b] [Sys91]. A dynamically variable crossover, whose rate depends on the spread of the population fitness values, has been reported by Booker [Boo87]. The advantage of this technique is that the operators are dynamically adapting to the data. A reduction in the spread indicates a converging population so the crossover rate is decreased to give the mutation operator more opportunity to find new variations.

Others have reported [Ack87, Bra91, Fog89, MJ91] increased performance by dynamically varying the mutation rate.

C5.9 Genetic Drift

In a multimodal problem the fitness function will have several peaks. A traditional GA will converge on only one peak even if the peaks have equal fitness. This migration toward a single peak may cause the

GA to converge away from some other possibly more fit peak. This bias toward a single peak is called genetic drift [GR87].

In order to thoroughly search all of the peaks, it is desirable to avoid genetic drift and create several sub-populations, each of which is converging toward a different peak. The sub-populations are analogous to biological species and the peaks they occupy are analogous to biological niches. Several modifications to the traditional GA have been proposed to accomplish this speciation of the population.

One approach to encouraging speciation is to maintain diversity. A technique introduced by Cavicchio [GR87], is to replace the parent only if the offspring's fitness exceeds that of the inferior parent. Under this selection method, strings tend to replace others which are similar to themselves, helping to prevent convergence on a single maximum. Stadnyk [Sta87] refined this technique further. Here the selection of individuals is modified such that new offspring are only allowed to replace others which are in the same niche and have low fitness.

Another approach to overcoming genetic drift was described by Beasley, Bull & Martin [BBM93]. They used multiple runs of the GA to find all of the peaks. After the first peak is found the fitness function is modified so that it will not be found again. The next run of the GA will, therefore, find a different peak. This continues until all of the peaks of interest have been found.

To encouraging speciation, Booker [Boo85] uses a scheme to restricted mating. Here only similar individuals are allowed to mate on the assumption that similar individuals will produce similar offspring. The

accuracy of this assumption depends on the epistasis inherent in the particular coding scheme used. Restricted mating encourages speciation and reduces the production of offspring that fall in the valley between the two maxima. These offspring are called "lethals" and are the undesirable product of parents from two different niches. Restricted mating is borrowed from nature where two individuals from different species can not produce offspring.

C5.10 Diploidy and Dominance

In certain situations, the objective function is not static but varies from generation to generation. In nature, for example, the climate will change over and time and competitors will come and go. These environmental changes directly affect the fitness of individuals to survive and prosper.

In this situation, it is advantageous to maintain a copy of previously used but no longer valid solutions just in case they are once again needed. This can be done, as it is in nature, through the use of a double string or diploid chromosome.

The GAs discussed so far have used a single string (or haploid) chromosome. A diploid chromosome actually has two strings with each gene location having two possible values. Which of these gene values is actually expressed is determined by dominance. The gene value which contributes the most to the organisms overall fitness will be set to dominate over the other gene value for this gene. If environmental conditions change, it is much easier to make the recessive gene value

dominant than to search for a new fitter gene value. This mechanism works best if the environment routinely switches between two states such as warm & ice-aged.

Diploidy in a GA might be useful for real time systems like a petroleum refinery where ambient temperature, humidity and barometric pressure gradually change over time and affect the optimum processing parameters. In spite of the occurrence of diploidy in nature, it is not widely seen in the GA world [Gol89a].

C5.11 Exploitation of Domain Knowledge

One of a GA's greatest strengths is that it can efficiently search for a solution with no more knowledge of the problem than that contained in the objective function. But as with any search technique, incorporating some domain specific knowledge into the design of the GA can greatly increase its performance. This has been accomplished by replacing crossover alone[SVG87], [Gre87] or both crossover and mutation [Dav91b], [Gol89a] with more task specific operators.

Another area where domain specific knowledge can easily be applied is in the generation of the initial population[Gre87], [SG90]. Normally this is randomly generated but a little advance knowledge can be used to great advantage by providing a "best guess" at to where likely solutions may be found. This jump start can greatly speed up the search

C5.12 Problem Constraints and Invalid Chromosomes

Many problems have a number of constraints giving rise to the unfortunate side effect that certain chromosome values may violate one or more of these constraints. It is also possible that the number of discrete solutions to the problem is not a power of 2 so it can not be expressed exactly as a binary number. In this case, the binary number will be larger than the number of available solutions resulting in a number of chromosome values that are not valid. A simple example of this is the problem to find the largest number between 1 and 100. A 6 bit binary chromosome going from 0 to 63 is not large enough to encode this while a 7 bit chromosome's range of 0 to 127 would have 28 invalid chromosome values.

The ideal solution is to use domain knowledge to prevent invalid chromosomes from being produced in the first place. But domain specific knowledge is not always available and requiring it reduces the robustness of a pure GA. But when it is available, it eliminates the time used to process and identify invalid values.

Without domain specific knowledge, there is no guarantee that such invalid codes will not arise. Crossover and mutation will explore the whole range of the chromosome space including valid and invalid areas. To deal with this, a number of solutions have been proposed [DeJ85].

The first and easiest is to simply assign an invalid chromosome a low fitness value. This makes the most sense in terms of the GA process, for indeed, such a chromosome is not fit at all. But processing a lot of low value chromosomes reduces the performance of the GA.

Another treatment for invalid chromosomes is to simply disregard them and produce a different, valid chromosome. This requires that all offspring be checked for validity after they are generated and discarding any that are found to be invalid. This may result in discarding fatal chromosomes which contain some great gene values intermixed among the fatal gene values.

A promising method for dealing with invalid chromosomes is chromosome remaping where invalid chromosomes are mapped onto valid ones.

C5.12.1 Chromosome Remaping

There are currently two types of chromosome remapping in use now. The first, fixed remapping , takes a particular invalid value and either changes it to some other particular valid value or processes it as if it were that other value. While the remapping mechanism is simple and it essentially removes all invalid chromosomes from the search space, it has the disadvantage that some problem solutions are represented by two chromosomes and others by just one. This causes the GA to search the dual chromosomed space more than the single chromosomed space.

Random remapping tries to fix this shortcoming by remapping an invalid value to a randomly chosen valid value. This eliminates the representational bias problem, but completely discards all parental inheritance information, instead opting to choose a random offspring when an invalid offspring is encountered.

C6. A GA implementation Using Java

The simple GA was implemented as a Web based Java applet. It uses only the three basic operators: roulette wheel selection, one point crossover and low-probability mutation and can only solve one problem. That problem is the search for the number X between 0 and 1 that will maximize the expression $f(X)=X^{10}$. The value X is coded as a 32 bit binary chromosome with 0000 0000 0000 0000 0000 0000 0000 0000 representing 0 and 1111 1111 1111 1111 1111 1111 1111 1111 representing 1.

Since the solution is known (X=1), the results of the GA can be plotted as percentage of the optimum value. For each generation the best solution and the average of the whole population are plotted. The simulation runs for 400 generations and typical values of mutation and crossover rate are used. In order to make such a simple problem hard for the GA, an unrealistically small population size of 6 individuals is used. In the example run shown in Figure 3, the algorithm got to within 98.8% of the correct answer in 180 generation. The remaining generation did not produce any further benefits. This is typical of GA since they are good at finding good quality solutions in a short time but not as good at finding the best solution.

The complete source code is included in Appendix E and a working version is available on the Web at:
http://www.stclairc.on.ca/people/pages/twilliams/ga/.

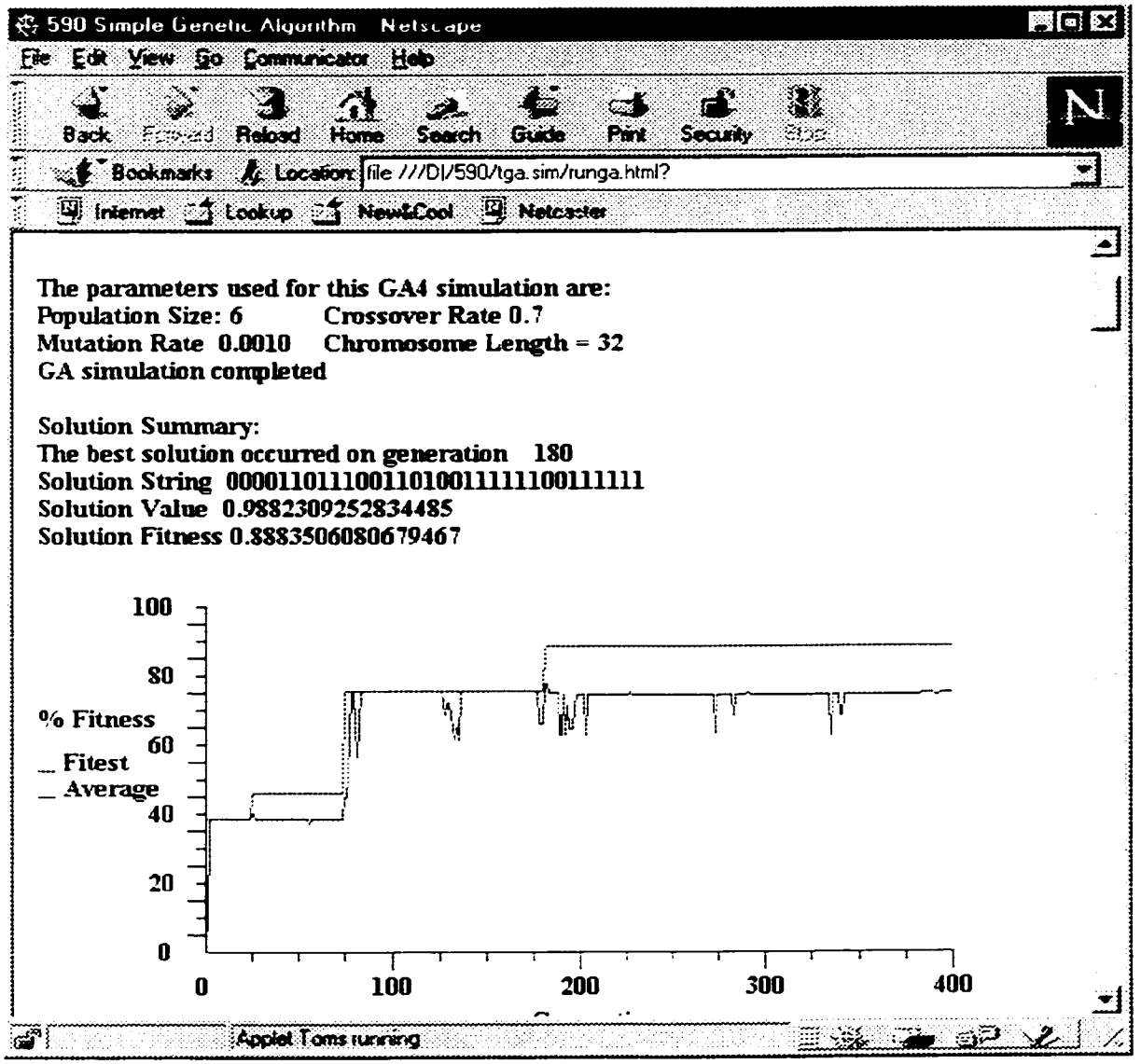

Figure 3 GA sample run from

http://www.stclairc.on.ca/people/pages/twilliams/ga/

C7. Future Trends

Much work still needs to be done to ascertain the optimum parameters for genetic algorithms. There are many interrelated variables such as population size, number of generations required, mutation rate, crossover rate and alphabet cardinality. The proper values to use in a given situation is not at all clear.

In many cases research results are based on empirical observations or techniques borrowed from biology with little or no theoretical justification. The theoretical understanding of GAs needs to be greatly expanded so that the inherently complex interactions can be better understood and new techniques can be proposed.

The difficulties encountered in problems with a high epistasis must be addressed. There are two avenues of approach to this issue; develop new techniques to reformulate a highly epistatic problem to one with lower epistasis or develop enhanced procedures that are capable of dealing with highly epistatic problems.

Much more work needs to be done on niche formation. Developing a GA to find all of the best maxima while avoiding the merely good maxima will require a great deal of progress in the both technique and understanding.

The study of GAs will be an important area of research for a long time to come.

C8. Conclusion

Genetic Algorithms are an important tool to be used against the problems encountered in search and optimization. It has properties that make it superior to other techniques for many broad classes of problems. It can work when there is absolutely no problem domain knowledge other than the objective function. This report has examined their fundamental principles and surveyed the current research in this area. While much progress has been made in using and understanding them it is clear that there is a lot of room for further improvements and enhancements.

Appendix D Source Code for the Example Problems

File maxones.cpp: The maximum ones objective function

```
//  A C++ Program
#include <process.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
main(int argc, char* argv[])  // access command-line arguments
{

      FILE *f;
      const int priorityLen = 5;
      char chromosome[90];
      float value;
      int i,j;


       // Read in Chromosome
       if (argc != 3 ) {
              exit(2);
       }
       strcpy (chromosome,argv[1]);
       printf("chrom = %s\n",chromosome);

       // count the number of 1's in the chromosome
       for ( i=0; chromosome[i] != '\0'; i++) {
              if (chromosome[i] == '1' ) {
                     //printf("I got a 1\n");
                     value +=1.0;
              }
              //else printf("I got a %c\n",chromosome[i]);
       }

       // Write results to a file
       if ((f = fopen(argv[2], "wt")) == NULL) {
              exit(3);
       }
       fprintf(f,"%f",value);
       fclose(f);
       //printf("%f 3= %f 4= %f",chromosome,i,1.234,j);
}
```

File maxonesd.cpp: The maximum ones chromosome decoder

```
//A C++ Program
#include <process.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <graphics.h>

main(int argc, char* argv[])  // access command-line arguments
{

      char chromosome[90];
      float value;
      int i,j;

        // Read in Chromosome
        if (argc != 2 ) {
                exit(2);
        }


        strcpy (chromosome,argv[1]);
        printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
       printf("Chromosome Decoder for the Max Ones problem\n");
       printf("-----------------------------------------\n");
       printf("The Chromosome is %s\n",chromosome);

       // count the number of 1's in the chromosome
       for ( i=0; chromosome[i] != '\0'; i++) {
               if (chromosome[i] == '1' ) {
                       //printf("I got a 1\n");
                       value +=1.0;
               }

       }
       printf("It has a fitness value of %f\n\n",value);

}
```

File xto10.cpp: The X to the 10th problem objective function

```
// A C++ Program
#include <process.h>
#include <stdio.h>
#include <math.h>
#include<string.h>
main(int argc, char* argv[])  // access command-line arguments
{
      double value = 0.0;
      FILE *f;
      char chromosome[90],buf[10];
      int i,digit, base =10;

        // Read in Chromosome
        if (argc != 3 ) {
              exit(2);
        }

        //sscanf(argv[1],"%lf",&chromosome);
        strcpy (chromosome,argv[1]);

        // Evaluate fitness of chromosome

        // convert string to a decimal number less than 1
         for (i=0;chromosome[i] != '\0';i++) {
                sprintf (buf,"%c",chromosome[i]);
                sscanf(buf,"%d",&digit);
                value += digit / pow(base,i+1);
         }

        // evaluate X to the 10th
        value = pow(value,10);

        // Write results to a file
        if ((f = fopen(argv[2], "wt")) == NULL) {
              exit(3);
        }
        fprintf(f,"%12.12g",value);
        printf("%12.12g",value);
        fclose(f);
        //printf("%f 3= %f 4= %f",chromosome,i,1.234,j);
}
```

File xto10.cpp: The X to the 10th Chromosome Decoder Function

```
// A C++ Program
#include <process.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <graphics.h>

main(int argc, char* argv[])  // access command-line arguments
{
      char chromosome[90],buf[90];
      double value;
      int i,digit, base =10;

        // Read in Chromosome
        if (argc != 2 ) {
```

```
        exit(2);
    }

    strcpy (chromosome,argv[1]);
    printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
    printf("Chromosome Decoder for the X to the 10th                    problem\n");
    printf("-----------------------------------------------
     \n");
    printf("The Chromosome is %s\n",chromosome);

      // convert string to a decimal number less than 1
      for (i=0;chromosome[i] != '\0';i++) {
            sprintf (buf,"%c",chromosome[i]);
            printf("printing %c to buf",chromosome[i]);
            sscanf(buf,"%d",&digit);
            printf("Got %d from buf\n",digit);
            value += digit / pow(base,i+1);
            printf("value is now %g\n",value);
      }


    printf("This represents the value %12.12g\n",value);
    value = pow(value,10);
    printf("It has a fitness value of %12.12g\n\n",value);


}
```

File tsp.cpp: The Traveling Salesman Problem Objective Function

```
// A C++ Program
#include <process.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
main(int argc, char* argv[])  // access command-line arguments
{
      const int numCities = 10;
      double value,x,y,dist;
      struct citytype {
           char binPriorty[numCities];
           double Priority;
           double locationX;
           double locationY;
      } ;
      double Xs[numCities] = {  0.25,0.873,0.368,0.519,0.893,0.465,-0.509,-0.837,-0.119 ,-
0.378};
      double Ys[numCities] = {  0.999,0.487,0.929,-0.854,-0.448,0.885,0.860,-0.546,-
0.993,0.925};
      FILE *f;
      const int priorityLen = 5;
      int temp, rank[numCities];
      char chromosome[90];
      citytype city[numCities];
      int i,j;
      for ( i=0; i<numCities ; i++) {
               rank[i] = i;
               city[i].locationX = Xs[i];
               city[i].locationY = Ys[i];
      }

       // Read in Chromosome
       if (argc != 3 ) {
              exit(2);
       }
       strcpy (chromosome,argv[1]);
       // Get priority of each city from chromosome.
       // Each 5 bits of chromosome is priorty of each city.
       for ( i=0; i<numCities ; i++) {
                city[i].Priority = 0;
                for ( j=0; j < priorityLen ; j++) {
                       if (chromosome[i*priorityLen + j] == '1' ) {
                              //Priority reversed i.e lsd first, msd last
                              city[i].Priority += pow (2,j);
                       }
               }
       }
       // Rank cities according to priority.
       for ( i=0; i<numCities -1 ; i++) {
               for ( j=i+1; j<numCities ; j++) {
                       if (city[rank[i]].Priority < city[rank[j]].Priority) {
                              temp = rank[j];
                              rank[j]=rank[i];
                              rank[i]=temp;
                       }
               }
       }
       // evaluate chromosome
       value = 0.0;
       for ( i=1; i<numCities ; i++) {
               x= city[rank[i]].locationX - city[rank[i-
        1]].locationX;
```

```
            y= city[rank[i]].locationY - city[rank[i-
        1]].locationY;
            dist = sqrt(x*x+y*y);
            value += dist;
        }

        // check for invalid chromosome
           // 10 cities must be unique
        for ( i=0; i<numCities-1 ; i++) {
                for ( j=i+1; j<numCities ; j++) {
                        if ( rank[i] == rank[j] )  {
                                value = 9999999.0;  // no good
                        }
                }
        }

        // This is a minimization problem so use neg fitness
        value = 0.0 - value;

        // Write results to a file
        if ((f = fopen(argv[2], "wt")) == NULL) {
                exit(3);
        }
        fprintf(f,"%f",value);
        fclose(f);
        //printf("%f 3= %f 4= %f",chromosome,i,1.234,j);
}
```

File xto10.cpp: The Traveling Salesman Problem Chromosome Decoder

```
Rem      A Visual Basic Program
Form1 - 1

Private Type citytype
        Priority As Double
        locationX As Double
        locationY As Double
        name As String
End Type
Dim numCities, i, j As Integer
Dim value, x, y, dist As Double
Dim temp, rank() As Integer
Dim city() As citytype
Private Sub Form_Paint()

 numCities = 10
 Xs = Array(0.25, 0.873, 0.368, 0.519, 0.893, 0.465, -0.509, -                    0.837, -
0.119, -0.378)
 Ys = Array(0.999, 0.487, 0.929, -0.854, -0.448, 0.885, 0.86, -                   0.546, -
0.993, 0.925)
 myName = Array("A0", "B1", "C2", "D3", "E4", "F5", "G6", "H7",                   "I8", "J9")
 Dim priorityLen  As Integer
 priorityLen = 5
 ReDim city(numCities)
 Dim chromosome As String
 ReDim rank(numCities)
 i = 0
 Do
     rank(i) = i
     city(i).locationX = Xs(i)
     city(i).locationY = Ys(i)
     city(i).name = myName(i)
     i = i + 1
 Loop While i < numCities
 tr0 = "00000"
 tr1 = "10000"
 tr2 = "01000"
 tr3 = "11000"
 tr4 = "00100"
 tr5 = "10100"
 tr6 = "01100"
 tr7 = "11100"
 tr8 = "00010"
 tr9 = "10010"


 ' Get chromosome from command line.
 chromosome = Command
 ' Default chromosome for testing.
 If chromosome = "" Then
     chromosome = tr3 & tr6 & tr4 & tr8 & tr7 & tr5 & tr1 & tr0                    &
tr9 & tr2
 End If
 ' Get priority of each city from chromosome.
 ' Each 5 bits of chromosome is priorty of each city.
 i = 0
 Do
     city(i).Priority = 0
     j = 0
     Do
```

```
        Bit = Mid(chromosome, i * priorityLen + j + 1, 1)
        If Bit = "1" Then
            Rem Priority reversed i.e lsd first, msd last
            city(i).Priority = city(i).Priority + 2 ^ j
        End If
        j = j + 1


Form1 - 2

      Loop While j < priorityLen
     i = i + 1
 Loop While i < numCities


  ' Rank cities according to priority.
 i = 0
 Do
    j = i + 1
    Do
        If city(rank(i)).Priority < city(rank(j)).Priority Then
            temp = rank(j)
            rank(j) = rank(i)
            rank(i) = temp
         End If
    j = j + 1
    Loop While j < numCities
 i = i + 1
 Loop While i < numCities - 1


  '   Plot them in order of ranking.
  '   scale factor = sf, X center = xc, Y center = yc
 sf = 2500
 xc = 3000
 yc = 2700
 radius = 100
 i = 1
 Do
     Circle (city(rank(i - 1)).locationX * sf + xc, city(rank(i             - 1)).locationY *
sf + yc), radius, QBColor(4)
     Print "    "; city(rank(i - 1)).name
     Line (city(rank(i - 1)).locationX * sf + xc, city(rank(i -             1)).locationY * sf
+ yc)-(city(rank(i)).locationX *             sf + xc, city(rank(i)).locationY * sf + yc)
     i = i + 1
 Loop While i < numCities
 Circle (city(rank(i - 1)).locationX * sf + xc, city(rank(i -             1)).locationY * sf
+ yc), radius, QBColor(4)
 Print "    "; city(rank(i - 1)).name
 CurrentX = 1000
 CurrentY = 5500
 Print "                  This order is: ";
 i = 0
 Do
    Print city(rank(i)).name; " - ";
    i = i + 1
 Loop While i < numCities - 1
 Print city(rank(i)).name

End Sub
```

1. Appendix E: Survey GA Program Listing

```
/*****************************************************************/
/* Simple Genetic Algorithm by Tom Williams
/* Finds the largest 32 bit number between 0 an 1 that maximizes
/* f(x) = x to the 10th power. It runs for 400 generations and plots
/* the population average and best individual for each generation
/* Some data structures are based on examples by Zbigniew
/* Michalewicz [Mic96]
/*****************************************************************/
import java.awt.*;
import java.lang.*;
import java.io.*;
import java.net.*;

public class tga extends java.applet.Applet {

        // set the GA values
        private static int    populationSize = 6;
        private static double         crossoverRate  = .7;
        private static double         mutationRate   = .001;
        private static int            numGenerations = 400;

        Image myBufImage;
        public Graphics myBufG;
        Font f = new Font("TimesRoman", Font.BOLD, 14);
        static String outline[] = new String[2150];
        static String outnum[] = new String[2150];
        static int numlines, oldGenBest, oldGenAvg, newGenBest, newGenAvg;
        static final int originX=90, originY=400;
        static double GenBest[]=new double[500];
        static double GenAvg[]=new double[500];
        private final Chromosome theBest = new Chromosome();
        private int theBestGeneration;
private Chromosome[] population    = new Chromosome[populationSize];
private Chromosome[] newPopulation = new Chromosome[populationSize];
        private int generationCount;     // generation counter
        private Selection theSelection = null;
        private Crossover theCrossover = null;

        public tga() { super(); }

public void myprint(String column1,String column2){
                outline[numlines] = column1;
                outnum[numlines++] = column2;
        }

        public void init() {
                myBufImage= createImage(700,600);
                myBufG = myBufImage.getGraphics();
                resize(750,600);
        }

        public void paint(Graphics g) {
                g.drawImage(myBufImage,0,0,this);
        }

        public void doText() {
                myBufG.setFont(f);
                myBufG.setColor(Color.black);
                for (int i=0;i < numlines;i++){
                        myBufG.drawString(outline[i], 1, 20+i*16);
                        myBufG.drawString(outnum[i], 150, 20+i*16);
```

```
        }
    }

    public void doAxis() {
        myBufG.setFont(f);
        myBufG.setColor(Color.black);
        myBufG.drawLine(originX,originY,originX+400,originY);
        myBufG.drawLine(originX,originY,originX,originY-200);

        for (int i=25; i <= 400; i += 25) {
            if (i%100 == 0)
             myBufG.drawLine(originX+i,originY,originX+i,originY+10);
            else
             myBufG.drawLine(originX+i,originY,originX+i,originY+5);
        }
        myBufG.drawString("0",originX-5,originY+25);
        myBufG.drawString("100",originX+90,originY+25);
        myBufG.drawString("200",originX+190,originY+25);
        myBufG.drawString("300",originX+290,originY+25);
        myBufG.drawString("400",originX+390,originY+25);
        for (int i=originY; i > originY -200; i -= 10 ) {
            if  (i%20 == 0)
                myBufG.drawLine(originX,i-10,originX-10,i-10);
            else
                myBufG.drawLine(originX,i-10,originX-5,i-10);
        }
        myBufG.drawString("0",originX-25,originY+5);
        myBufG.drawString("20",originX-30,originY-35);
        myBufG.drawString("40",originX-30,originY-75);
        myBufG.drawString("60",originX-30,originY-115);
        myBufG.drawString("80",originX-30,originY-155);
        myBufG.drawString("100",originX-38,originY-195);
        myBufG.drawString("% Fitness",1,originY-130);
        myBufG.setColor(Color.red);
        myBufG.drawLine(1,originY-105,10,originY-105);
        myBufG.setColor(Color.black);
        myBufG.drawString(" Fitest",11,originY-105);
        myBufG.setColor(Color.blue);
        myBufG.drawLine(1,originY-90,10,originY-90);
        myBufG.setColor(Color.black);
        myBufG.drawString(" Average",11 ,originY-90);
        myBufG.drawString(" Generation ",originX+170,originY+45);

    }

    public void doPlot() {
        GenBest[0] *= 200;
        GenAvg[0] *= 200;
        oldGenBest = (int)GenBest[0];
        oldGenAvg = (int)GenAvg[0];
        for (int i=1; i <  numGenerations-1; i++ ) {
             GenBest[i] *= 200;
        GenAvg[i] *= 200;
        newGenBest = (int)GenBest[i];
        newGenAvg = (int)GenAvg[i];
                myBufG.setColor(Color.blue);
myBufG.drawLine(originX+i,originY-oldGenAvg,originX+i+1,originY-newGenAvg);
                myBufG.setColor(Color.red);
myBufG.drawLine(originX+i,originY-oldGenBest,originX+i+1,originY-newGenBest);
        oldGenBest=newGenBest; oldGenAvg=newGenAvg;
        }
    }
```

```
        public void start() {

                numlines=0;
                // Print out the parameters
                myprint ("The parameters used for this GA4 simulation are:"," ");
                myprint ("Population Size: " + populationSize ," Crossover Rate "+ crossoverRate
);
                myprint  ("Mutation Rate  "+ mutationRate ," Chromosome Length = "+
Chromosome.getChromosomeLength() );
                new Chromosome(); // force loading class Chromosome and setting static variables
                doAxis();
                doGA();
                doPlot();
                repaint();
                doText();   // print text
        }

        private void doGA() {
                //      initialize();
                generationCount= 0;
                theSelection = new Selection();
                theCrossover = new Crossover();

                for (int j = 0; j < populationSize; j++) {
                        population[j] = new Chromosome();
                        population[j].initializeChromosomeRandom();
                        newPopulation[j] = new Chromosome();
                }

                //  Search the population for the individual with the highest fitness
                int BestSoFarIndex = 0;       // index of the current best individual
                double BestSoFar = population[BestSoFarIndex].getFitness();
                double fitness;
                for (int i = 1; i < populationSize; i++) {
                        if ((fitness = population[i].getFitness()) > BestSoFar ) {
                                BestSoFarIndex = i;
                                BestSoFar = fitness;
                        }
                }

                // once the best member in the population is found, copy the genes
                population[BestSoFarIndex].copyChromosome(theBest);
                UpdateGenerationStats();
                while (generationCount <= numGenerations) {
                        generationCount++;
                        theSelection.select(population, newPopulation, populationSize);
                        //swapPopulationArrays();
                        Chromosome[] temp = population;
                        population = newPopulation;
                        newPopulation = temp;


                        // Perform Crossover
                        int parent1 = 0, chosenCount  =  0;
                        for (int i = 0; i < populationSize; ++i) {
                        if (MyRandom.dbl() < crossoverRate) {
                                ++chosenCount;
                                if (chosenCount % 2 == 0) {
                                        theCrossover.xOver(population[parent1],
                                        population[i]);
                                } else parent1 = i;
```

```
                }
            }

            // Perform Mutation
            int chromosomeLength = Chromosome.getChromosomeLength();
            for (int i = 0; i < populationSize; i++) {
                for (int j = 0; j < chromosomeLength; j++) {
                        if (MyRandom.dbl() < mutationRate) {
                            population[i].mutateGene(j);
                        }
                    }
            }


            double currentBestFitness = population[0].getFitness();
            double next = -1;
            int currentBest = 0; // index of the current best individual

            for (int j = 1; j < populationSize; j++) {
                if ((next = population[j].getFitness()) > currentBestFitness ) {
                        currentBest = j;
                        currentBestFitness = next;
                }
            }
            if (currentBestFitness > theBest.getFitness()) {
                        population[currentBest].copyChromosome(theBest);
                        theBestGeneration = generationCount;
                }
            UpdateGenerationStats();

            }
            // done so print findings
            myprint("GA simulation completed", "");
            myprint(" "," ");
            myprint("Solution Summary: "," ");
            myprint("The best solution occurred on generation" , " " +
            theBestGeneration  );
            myprint("Solution String  " + theBest.toGenotype() , " ");
            myprint("Solution Value  " + theBest.phenotype(), " ");
            myprint("Solution Fitness " + theBest.getFitness()," ");
        }


        private void UpdateGenerationStats( ) {
                double best = 0.0;    // best fitness of this generation
                double average;       // average  fitness of this generation
                double total = 0.0;  // total fitness of this generation
                double fitness;

                for (int i = 0; i < populationSize; i++) {
                        fitness = population[i].getFitness();
                    if (best < fitness) best = fitness;
                        total += fitness;
                }
                average = total/(double)populationSize;
                best = theBest.getFitness();
                GenBest[generationCount]=best;
                GenAvg[generationCount]=average;
        }
}

/*****************************************************************************/
/* Cromosome Class
```

```
/* Defines the structure and operators for fixed length binary chromosomes
/*************************************************************************/
public class Chromosome {
        protected boolean[] gene = null;
        protected final static  int  chromosomeLength = 32;
        protected double  fitness, propFitness, accumFitness;

        protected Chromosome() {
                gene = new boolean[chromosomeLength];
        }

        public static final int getChromosomeLength() {
                return chromosomeLength;
        }

        public final double getFitness() {
                if (fitness < 0)    {
                        fitness = evalChromosome();
                }
                return fitness;
        }

        public  String toString() {
                return this.toGenotype();
        }

        public final double propFitnessGet() {
                 return propFitness;
        }

        public final double accumFitnessGet() {
                 return accumFitness;
        }

        public final void propFitnessPut(double value) {
                 propFitness = value;
        }

        public final void accumFitnessPut(double value) {
                 accumFitness = value;
        }

        public void copyChromosome(Chromosome newCopy) {
                newCopy.fitness = this.fitness;
                newCopy.propFitness = this.propFitness;  // really belong
                newCopy.accumFitness = this.accumFitness;  // in the superclass
                for (int i = 0; i < chromosomeLength; i++) {
                        newCopy.gene[i] = this.gene[i];
                }
        }

        public void initializeChromosomeRandom() {
                for (int i = 0; i <chromosomeLength; i++) {
                        gene[i] = MyRandom.bool();
                }
                fitness = -1;           // -1 signifies that fitness is not set
                propFitness = 0; accumFitness = 0;
        }

        public void clearChromosome() {
                for (int i = 0; i <chromosomeLength; i++)     {
                        gene[i] = false;
                }
                fitness = -1;
```

```
            propFitness = accumFitness = 0;
        }

        public Number getGene(int i) {
            return (Number) (gene[i]?new Integer(1):new Integer(0));
        }

        public void setGene(int i, Number n) {
            gene[i] = ((Integer) n).intValue() != 0;
            fitness = -1;
            propFitness = 0; accumFitness = 0;
        }

        public void mutateGene(int i) {
            gene[i] = !gene[i];
            fitness = -1;    // fitness unknown
            propFitness = accumFitness = 0;
        }

        public String toGenotype() {
            String genotype = "";
            for (int i = 0; i < chromosomeLength; i++) {
                        genotype += (gene[i] ? "1" : "0");
            }
            return genotype;
        }

        // Fitness Function is Chromosome value to the 10th power
        protected double evalChromosome(){
            return Math.pow(phenotype(), (double) 10);
        }

        double phenotype() {
            double value = 0;
            for (int i = 0; i < chromosomeLength; i++) {
                        if (gene[i])  {
                        value += Math.pow((double)2.0, (double)(i));
                    }
             }
            return value/( Math.pow((double)2.0, (double)chromosomeLength )- 1);
        }

}




public class MyRandom {
         // Return a random double in [0,1).
        public static double dbl() {
            return Math.random();
        }

        // Return a random boolean (false or true).
        public static boolean bool() {
            return Math.random() >= 0.5;
        }

        // Return a random integer from 1 to n inclusive.
        public static int integer(int n) {
            return (int)(Math.random()*n + 1);
        }
```

```
}


/*****************************************************************/
/* Crossover Operator
/* Single point crossover of the two selected parents.
/* Offsprings replace parent1 & parent2
/*****************************************************************/
public class Crossover {
        public void xOver(Chromosome parent1, Chromosome parent2) {
                int point; // crossover point
                int chromosomeLength = Chromosome.getChromosomeLength();
                Number swapTmp;
                point = MyRandom.integer(chromosomeLength-1);
                for (int i = 0; i < point; i++) {
                        swapTmp = parent1.getGene(i);
                        parent1.setGene(i, parent2.getGene(i));
                        parent2.setGene(i, swapTmp);
                }
         }
}

/*****************************************************************/
/* Selection Operator
/* Select parents for mating using the "roulette wheel: method
/*****************************************************************/
 public class Selection {
        public void select(Chromosome[] individual, Chromosome[] newIndividual,
        int populationSize){
                // the population is represented as an array
                // of individuals i.e. individual[]
                double p, sum = 0;
                int i,j;

                // need the population fitness
                for (i = 0; i < populationSize; i++) {
                        sum += individual[i].getFitness();
                }

                // calculate each individual's proportion of total fitness
                for (i = 0; i < populationSize; i++) {
                        individual[i].propFitnessPut(individual[i].getFitness()/sum);
                }

                individual[0].accumFitnessPut(individual[0].propFitnessGet());

                // acccumulate the relative fitnesses from 0 to 1
                for (i = 1; i < populationSize; i++) {
                                individual[i].accumFitnessPut(individual[i-1].accumFitnessGet() +
individual[i].propFitnessGet());
                }

                // Use the cumulative fitness to select  new population.
                for (i = 0; i < populationSize; i++) {
                                p = MyRandom.dbl();
                                if (p < individual[0].accumFitnessGet())  {
                                        individual[0].copyChromosome(newIndividual[i]);
                                }
                                else {
                                        for ( j = 0; j < populationSize; j++) {
                                            if (p >= individual[j].accumFitnessGet()
                                                && p < individual[j+1].accumFitnessGet()) {
                                                individual[j+1].copyChromosome(newIndividual[i]);
```

```
                                    }
                                }
                            }
                        }
                    }
                }
```

Appendix F Source Code for the GA testbed

File testbed.java The Console Window

```
import java.awt.*;
import java.lang.*;
import java.io.*;
import java.net.*;
import java.awt.event.*;
import layout.*;

public class testbed extends Frame implements ActionListener{

        // GA parameters
        boolean elitism = true;
        int xoverType = 0; // 0 = uniform, 1 = 1 point, 2=2 point
        int popSize,chromCard,chromLength,NumGens;
        double mutRate,crossRate;
        String fitFunc, chromDecode;
        int gacount =0;

        //GUI Objects
        TextField popBox, mrBox, fitFuncBox,chromFuncBox,
        cromCardBox, cromLenBox;
        TextField xoverRateBox,genBox;
        Button b1,b2;
        Choice xover;
        Checkbox eliteOn, eliteOff;
        CheckboxGroup eliteGroup;
        Label errMessage;

        public static void main(String args[]) {
                testbed tb = new testbed();
                tb.show();  // will call paint()
                tb.setSize(520,260);
        }

        public testbed() {
                super("Testbed Console");
                MenuBar mb = new MenuBar();

                // add menus
                Menu file = new Menu("File");
                MenuItem miExit = new MenuItem("Exit");
                file.add(miExit);
                miExit.addActionListener(this);
                mb.add(file);
                setMenuBar(mb);

                // Create and set a position layout
                PositionLayout posLayout = new PositionLayout();
                setLayout(posLayout);
                int col1=10,col2=165,col3=280,col4=360;
                int row1=10,row2=40,row3=70,row4=100,row5=130;
                int row6=160,row7=180;

                // Run button
                PositionConstraints pCons;
                pCons = new PositionConstraints();
                pCons.x = col3 - 30;
                pCons.y = row7;
```

```
b1 = new Button("Run New GA");
posLayout.setConstraints(b1, pCons);
add(b1);

//Quit button
pCons = new PositionConstraints();
pCons.x = col4 + 30;
pCons.y = row7;
b2 = new Button("Quit");
posLayout.setConstraints(b2, pCons);
add(b2);
b1.addActionListener(this);
b2.addActionListener(this);

// Population Size text field
pCons = new PositionConstraints();
pCons.x = col1; pCons.y = row1;
Label  popLabel     = new Label ("Population Size");
posLayout.setConstraints(popLabel, pCons);
add(popLabel);

// Mutation Rate text field
pCons = new PositionConstraints();
pCons.x = col1;pCons.y =row2;
Label  mrLabel  = new Label ("Mutation Rate");
posLayout.setConstraints(mrLabel, pCons);
add(mrLabel);

// Fitness Function text field
pCons = new PositionConstraints();
pCons.x = col1;pCons.y =row3;
Label  fitLabel = new Label ("Fitness Function");
posLayout.setConstraints(fitLabel, pCons);
add(fitLabel);

// Chromosome Decoder Function text field
pCons = new PositionConstraints();
pCons.x = col1;pCons.y =row4;
Label  decoderLabel = new Label ("Chromosome Decoder");
posLayout.setConstraints(decoderLabel, pCons);
add(decoderLabel);

// Chromosome Cardinality text field
pCons = new PositionConstraints();
pCons.x = col1;pCons.y =row5;
Label  cromCardLabel = new Label ("Chromosome
Cardinality");
posLayout.setConstraints(cromCardLabel, pCons);
add(cromCardLabel);

// Chromosome Length text field
pCons = new PositionConstraints();
pCons.x = col1;pCons.y =row6;
Label  cromLenLabel = new Label ("Chromosome Length");
posLayout.setConstraints(cromLenLabel, pCons);
add(cromLenLabel);

// Population text box
pCons = new PositionConstraints();
pCons.x = col2;pCons.y =row1;
popBox = new TextField("14",5);
posLayout.setConstraints(popBox, pCons);
add(popBox);
```

```
// Mutation Ratetext box
pCons = new PositionConstraints();
pCons.x = col2;pCons.y =row2;
mrBox = new TextField(".01",5);
posLayout.setConstraints(mrBox, pCons);
add(mrBox);

// Fit Function text box
pCons = new PositionConstraints();
pCons.x = col2;pCons.y =row3;
fitFuncBox = new TextField("default.exe",10);
posLayout.setConstraints(fitFuncBox, pCons);
add(fitFuncBox);

// Chromosome Decoder Function text box
pCons = new PositionConstraints();
pCons.x = col2;pCons.y =row4;
chromFuncBox = new TextField("none",10);
posLayout.setConstraints(chromFuncBox, pCons);
add(chromFuncBox);

// Chrom Card text box
pCons = new PositionConstraints();
pCons.x = col2;pCons.y =row5;
cromCardBox = new TextField("2",3);
posLayout.setConstraints(cromCardBox, pCons);
add(cromCardBox);

// Chrom Length text box
pCons = new PositionConstraints();
pCons.x = col2;pCons.y =row6;
cromLenBox = new TextField("50",3);
posLayout.setConstraints(cromLenBox, pCons);
add(cromLenBox);

// Crossover Selection List
pCons = new PositionConstraints();
pCons.x = col3;pCons.y =row1;
xover = new Choice();
xover.addItem("uniform");
xover.addItem("1 point");
xover.addItem("2 point");
posLayout.setConstraints(xover, pCons);
add(xover);

// Crossover type text field
pCons = new PositionConstraints();
pCons.x = col4;pCons.y =row1;
Label  xoTypeLabel  = new Label ("Crossover Type");
posLayout.setConstraints(xoTypeLabel , pCons);
add(xoTypeLabel);

// Crossover Rate text box
pCons = new PositionConstraints();
pCons.x = col3;pCons.y =row2;
xoverRateBox = new TextField("0.9",5);
posLayout.setConstraints(xoverRateBox, pCons);
add(xoverRateBox);

// Crossover Rate text field
pCons = new PositionConstraints();
pCons.x = col4;pCons.y =row2;
Label  xoRateLabel  = new Label ("Crossover Rate");
posLayout.setConstraints(xoRateLabel , pCons);
```

```
                add(xoRateLabel );

                // Number Of Gens Box
                pCons = new PositionConstraints();
                pCons.x = col3;pCons.y =row3;
                genBox = new TextField("0",5);
                posLayout.setConstraints(genBox, pCons);
                add(genBox);

                // Number of Gens text field
                pCons = new PositionConstraints();
                pCons.x = col4;pCons.y =row3;
                Label  genLabel  = new Label ("Number Of Generations");
                posLayout.setConstraints(genLabel , pCons);
                add(genLabel );

                // Number of Gens text field 2
                pCons = new PositionConstraints();
                pCons.x = col4;pCons.y =row3 +20;
                Label  gen2Label  = new Label ("(Enter 0 for no limit)");
                posLayout.setConstraints(gen2Label , pCons);
                add(gen2Label );

                // Elitism radio buttons
                pCons = new PositionConstraints();
                pCons.x = col3;pCons.y =row5;
                eliteGroup = new CheckboxGroup();
                eliteOn = new Checkbox("Use Elitism",eliteGroup,true);
                eliteOn.setState(true);
                posLayout.setConstraints(eliteOn, pCons);
                add(eliteOn);
                pCons = new PositionConstraints();
                //pCons.x = col3;pCons.y =row5 + 18;
                pCons.x = col4 + 10;pCons.y =row5;
                eliteOff = new Checkbox("No Elitism",eliteGroup,false);
                posLayout.setConstraints(eliteOff , pCons);
                add(eliteOff );

                // Elitism text field
                pCons = new PositionConstraints();
                pCons.x = col3;pCons.y =row7;
                errMessage = new Label ("                                                                                ");
                errMessage.setForeground(Color.red);
                posLayout.setConstraints(errMessage, pCons);
                add(errMessage);
        }

        public void paint(Graphics g) {
        }

        public void actionPerformed(ActionEvent event) {
                if (event.getSource() == b1) {
                boolean error = false;
                errMessage.setText("                                    ");
                    try {
                //convert text box strings to numeric equivalents
                        popSize = Integer.parseInt(popBox.getText());
                        mutRate =
Double.valueOf(mrBox.getText()).doubleValue();
                        crossRate = Double.valueOf(xoverRateBox.getText()).doubleValue();
                        chromCard = Integer.parseInt(cromCardBox.getText());
                        chromLength = Integer.parseInt(cromLenBox.getText());
                        NumGens = Integer.parseInt(genBox.getText());
                        fitFunc = fitFuncBox.getText();
```

```
                chromDecode = chromFuncBox.getText();
            } catch (NumberFormatException e) {
                error = true;
                errMessage.setText("Entry Error. Please correct the typo.");
              }
                xoverType = xover.getSelectedIndex();
                if (eliteGroup.getSelectedCheckbox() == eliteOn )
                        elitism = true;
                else
                        elitism = false;
                if (!error) {
                        //Run a new GA
                        if (gacount == 0 ) {
                          Plotter ga0 =  new Plotter( elitism, xoverType,
                                popSize,chromCard,chromLength,NumGens,
                                mutRate, crossRate,fitFunc,chromDecode);
                        }
                        if (gacount == 1 ) {
                          Plotter ga1 =  new Plotter( elitism, xoverType,
                                popSize,chromCard,chromLength,NumGens,
                                mutRate, crossRate,fitFunc,chromDecode);
                        }
                        if (gacount == 2 ) {
                          Plotter ga2 =  new Plotter( elitism, xoverType,
                                popSize,chromCard,chromLength,NumGens,
                                mutRate, crossRate,fitFunc,chromDecode);
                        }
                        if (gacount == 3 ) {
                          Plotter ga3 =  new Plotter( elitism, xoverType,
                                popSize,chromCard,chromLength,NumGens,
                                mutRate, crossRate,fitFunc,chromDecode);
                        }
                        if (gacount == 4 ) {
                          Plotter ga4 =  new Plotter( elitism, xoverType,
                                popSize,chromCard,chromLength,NumGens,
                                mutRate, crossRate,fitFunc,chromDecode);
                        }
                        if (gacount == 5 ) {
                          Plotter ga5 =  new Plotter( elitism, xoverType,
                                popSize,chromCard,chromLength,NumGens,
                                mutRate, crossRate,fitFunc,chromDecode);
                        }
                        gacount++;
                }
        } else if (event.getSource() == b2) {
                // Quit Button pressed so exit
                System.exit(0);
        }
        if    (event.getSource() instanceof MenuItem) {
                String menupick =  ( (MenuItem)event.getSource()).getLabel()   ;
                if (    "Exit".equals(menupick)    ) {
                        //System.out.println("Exiting");
                        // dispose of ga frame dispose();
                        System.exit(0);
                        return;
                }
                if ( "Help1".equals(menupick)    ) {
                        HelpWin hw = new HelpWin(true);
                }
                if (    "About".equals(menupick)    ) {
                        HelpWin hw = new HelpWin(false);
                }
        }
}
```

```
}
```

File GAPlotter.java The Results Window

```
import java.awt.*;
import java.lang.*;
import java.io.*;
import java.net.*;
import java.awt.event.*;
import ptplot.*;

public class GAPlotter extends PlotLive implements ActionListener {

        // Initial GA parameters
        private boolean elitism = true;
        private int xoverType;
        private int populationSize, chromCard, chromLength,
        numGenerations;
        private double mutationRate,crossoverRate;
        private String fitFunc, chromDecode;

        private Chromosome theBest; //= new Chromosome();
        private int            theBestGeneration;     // generation where theBest first showed
up
        //      0..popSize-1 holds population and theBest holds the most fit
        private Chromosome[] population;
        private Chromosome[] newPopulation;
        private int            generationCount = 0;     // generation counter
        private Selection      theSelection;
        private Crossover      theCrossover;
        private int Xrange = 10;
        private double Yupper = 0.1;
        private double Ylower = 0.0;
        //private double _count = 0.0;

        Frame f;
        Button bclose, bpause, bdecode;
        boolean pause = false;
        boolean done = false;
        Label lab1,lab2,lab3;
        Panel pbottom,p1,p2;

        //public Plotter( int ps, double mr, int ng)  {
        public Plotter(boolean e, int xot, int ps, int chc, int chl, int ng,
                                    double mr, double xor,String ff, String chd)  {
                elitism = e;
                xoverType = xot;
                populationSize = ps;
                chromCard = chc;
                chromLength = chl;
                numGenerations = ng;
                mutationRate = mr;
                crossoverRate = xor;
                fitFunc = ff;
                chromDecode = chd;
                populationSize = ps;
                mutationRate = mr;
                numGenerations = ng;

                // Set up and initialize the GA structures
                theBest = new Chromosome(chromCard,chromLength,fitFunc);
                population    = new Chromosome[populationSize];
                newPopulation = new Chromosome[populationSize];
                theSelection = new Selection();
                theCrossover = new Crossover(chromLength, xoverType);
```

```
        //Gui components
        lab1 = new Label();
        lab2 = new Label();
        lab3 = new Label();
        bpause = new Button(" Pause GA ");
        bclose = new Button("Close");
        bdecode = new Button("Decode Best");

        Label whiteSpace1 = new Label("");
        Label whiteSpace2 = new Label("");
        p1 = new Panel();
        p1.setLayout(new GridLayout(1,5,8,8));
        p1.add(whiteSpace1);
        p1.add(bpause);
        p1.add(bclose);
        p1.add(bdecode);
        p1.add(whiteSpace2);

        pbottom = new Panel();
        pbottom.setLayout(new GridLayout(4,1,0,0));
        pbottom.add(p1);
        pbottom.add(lab1);
        pbottom.add(lab2);
        pbottom.add(lab3);

        Color lab2colour = new Color(0,0,128);
        lab1.setBackground(Color.white);
        lab2.setBackground(lab2colour);
        lab2.setForeground(Color.white);
        lab3.setForeground(Color.white);
        lab3.setBackground(lab2colour);
        f=new Frame("GA with population of " + populationSize +
                " using function "+fitFunc );
        String eliteStatus = "off"; if (elitism) eliteStatus = "on";
        lab1.setText(" Mutation Rate=" + mutationRate+ "    Crossover Rate="
                + crossoverRate + "     Elitism is " + eliteStatus);
        f.setLayout(new BorderLayout());
        f.setBackground(Color.lightGray);
        f.add("North",this);
        f.add("South",pbottom);
        bpause.addActionListener(this);
        bclose.addActionListener(this);
        bdecode.addActionListener(this);
        f.pack();
        f.show();
        repaint();
        f.setSize(430,450);  // size of whole window
        this.setSize(400,320);  // size of this plotLive on the frame f
        this.init();
        bpause.setSize(5,5);
        bclose.setSize(5,5);
        addLegend(0,"Best");
        addLegend(1,"Avg.");
        setXLabel("Generation");
        setYLabel("Fitness");
        this.start();

}

public void actionPerformed  (ActionEvent event) {
        if (event.getSource() == bpause) {
                if (pause == true ) {
                    if (!done) {
```

```
                        pause = false;
                        bpause.setLabel("Pause GA");
                    }
                }else {
                    if (!done) {
                        pause = true;
                        bpause.setLabel("Resume GA");
                    }
                }
            }
            if (event.getSource() == bclose) {
                pause = true;
                done = true;
                System.out.println("Exiting");
                f.dispose();

            }
            if (event.getSource() == bdecode) {
               if ( !chromDecode.equals("none") ){
                    //System.out.println("Inside bdecode cromDecode = %" + chromDecode +
"%");
                    CallNativeGa p = new CallNativeGa();
                    String uniqId = "decode";
                    String input = p.runObjFunc(chromDecode,theBest.toGenotype(),uniqId);
               }

            }
        }


        public  void addPoints() {
           if (!done) {
                for (int j = 0; j < populationSize; j++) {
                    population[j] = new Chromosome(chromCard,chromLength,fitFunc);
                    population[j].initializeChromosomeRandom();
                    newPopulation[j] = new Chromosome(chromCard,chromLength,fitFunc);
                }

                //  Search the population for the individual with the highest fitness
                int BestSoFarIndex = 0;       // index of the current best individual
                double BestSoFar = population[BestSoFarIndex].getFitness();
                double fitness;
                for (int i = 1; i < populationSize; i++) {
                    if ((fitness = population[i].getFitness()) > BestSoFar ) {
                        BestSoFarIndex = i;
                                BestSoFar = fitness;
                        }
                }

                // once the best member in the population is found, copy the genes
                population[BestSoFarIndex].copyChromosome(theBest);

                // MAIN GA Loop
                while (generationCount <= numGenerations || numGenerations == 0) {
                    while (pause);  //  wait in loop
                    genPlot();
                    generationCount++;
                    theSelection.select(population, newPopulation, populationSize);
                    //System.err.println("gen  " + Generation one.toGenotype() + " with " +
two.toGenotype());
                    //swapPopulationArrays();     swap po
                    Chromosome[] temp = population;
                    population = newPopulation;
```

```
                    newPopulation = temp;

                    // Crossover;
                    int one = -1;
                    int first  =  0; // count of the number of members chosen
                    for (int i = 0; i < populationSize; ++i) {
                        if (MyRandom.dbl() < crossoverRate) {
                            ++first;
                            if (first % 2 == 0) {
                                theCrossover.xOver(population[one],
population[i]);
                            } else one = i;
                        }
                    }
                    if (elitism) {
                    // keep the best by replacing one individual
                    theBest.copyChromosome(population[0]);
                }
                    //  Mutation
                    for (int i = 0; i < populationSize; i++) {
                        for (int j = 0; j < chromLength; j++) {
                            if (MyRandom.dbl() < mutationRate) {
                                population[i].mutateGene(j);
                            }
                        }
                    }

                    // update The Best    //         UpdateTheBest();
                    double currentBestFitness = population[0].getFitness();
                    double next = -1;
                    int currentBest = 0; // index of the current best individual

                    for (int j = 1; j < populationSize; j++) {
                        if ((next = population[j].getFitness()) > currentBestFitness ) {
                            currentBest = j;
                            currentBestFitness = next;
                        }
                    }
                    if (currentBestFitness > theBest.getFitness()) {
                        population[currentBest].copyChromosome(theBest);
                        theBestGeneration = generationCount;
                    }


                }
                // done so print findings
                genPlot();
                printChromosome("Best member (generation="+theBestGeneration+")", theBest);
                System.out.println("Done Whole GA Loop");
                done = true;
                bpause.setLabel("         ");
            }
        }


        private void genPlot() {
                double bestFitness=0.0;     // best population fitness
                double avg;                 // avg population fitness
                double sum=0.0;             // total population fitness
                double fitness;
                double data[] = new double[2];
                boolean connectDots = true;
                for (int i = 0; i < populationSize; i++) {
```

```
                        fitness = population[i].getFitness();
                        //myprint("Gen =" + generationCount  + "  chrom=" + i , "   f="  + "
val= " + population[i]. toPhenotype() //+ "   bits=" + population[i]. toGenotype()    ) ;
                        if (bestFitness < fitness) bestFitness = fitness;
                        sum += fitness;
                }

                avg = sum/(double)populationSize;
                bestFitness = theBest.getFitness();
                //GenBest[generationCount]=bestFitness;
                //GenAvg[generationCount]=avg;
                //System.err.println("Gen =" + generationCount  + "  bestFitness" + bestFitness
+   " GenAvg =" + avg     );

                // set axis for plot
                if (generationCount > Xrange ) {
                        Xrange*=2;
                        setXRange(0,Xrange);
                        repaint();
                }
                while (bestFitness > Yupper ) {
                        Yupper*=2;
                        setYRange(Ylower,Yupper);
                        repaint();
                //System.out.println("Reseting Ymax to: " + Yupper);
                }
                while (bestFitness < Ylower ) {
                        if (Ylower == 0 ) Ylower = -1;
                        else Ylower*=2;
                        setYRange(Ylower,Yupper);
                        repaint();
                //System.out.println("Reseting Ymin to: " + Ylower);
                }
                while (avg < Ylower ) {
                        if (Ylower == 0 ) Ylower = -1;
                        else Ylower*=2;
                        setYRange(Ylower,Yupper);
                        repaint();
                //System.out.println("Reseting Ymin to: " + Ylower);
                }
                if (generationCount == 0 ) connectDots = false;
                addPoint(0, generationCount,bestFitness,connectDots);
                addPoint(1, generationCount,avg,connectDots);

                lab2.setText("   Best fitness so far is " +  bestFitness + " from
chromosome:");
                lab3.setText("    " + theBest.toGenotype() );

        }

        private void printChromosome(String name, Chromosome c) {
                // System.out.println("Got inside printChromp");
        }

        public void init() {
                if (_debug >8 ) System.out.println("PlotLiveDemo: init");
                        setTitle("Genetic Algorithm Results");
                setYRange(Ylower,Yupper);
                setXRange(0,Xrange);
                setNumSets(4);
                        setMarksStyle("none");
                // Give the user direct control over filling graph.
                //makeButtons();
                setPlotting(true);
```

```
                super.init();
        }

}
```

File Chromosome.java

```
public class Chromosome { // designed for genes that are class Number
        static int idCount =0;
        private int[] gene;
        private int cardinality, Length;
        private double  fitness;
        private String obFunction;
        // relative fitness  = fitness / sum(population fitness)
        private double  rfitness;
        private boolean fitnessEvaluated;

        // cumulative relative fitness (from chrom[0] to this chrom) for roulette wheel
        private double  cfitness;

        public double rfitnessGet() {
                return rfitness;
        }
        public double cfitnessGet() {
                 return cfitness;
        }
        public void rfitnessSet(double value) {
                rfitness = value;
        }
        public void cfitnessSet(double value) {
                cfitness = value;
        }
        public  String toString() {
                return this.toGenotype();
        }

        public Chromosome(int cc,int cr,String ff ) {
                cardinality = cc;
                Length = cr;
                obFunction = ff;
                gene = new int[Length];
                fitnessEvaluated = false;
                fitness = rfitness = cfitness = 0;
        }

        public int getChromosomeLength() {
                return Length;
        }

        public final double getFitness() {
                if (!fitnessEvaluated)    {
                        fitness = evalChromosome();
                }

                return fitness;
        }


        public void copyChromosome(Chromosome c) { // copy this to c
                Chromosome destination = (Chromosome) c;
                destination.fitness  = this.fitness;
                destination.rfitness = this.rfitness;
                destination.cfitness = this.cfitness;
                fitnessEvaluated = this.fitnessEvaluated;
                for (int  locus = 0;  locus < Length;  locus++) {
                        destination.gene[ locus] = this.gene[ locus];
                }
        }
```

```
        public Chromosome cloneChromosome() {
                Chromosome theClone = new Chromosome(cardinality,Length,obFunction);
                copyChromosome((Chromosome) theClone);
                return (Chromosome) theClone;
        }

        public void initializeChromosomeRandom() {
                for (int locus = 0; locus < Length; locus++) {
                                gene[locus] = MyRandom.integer(cardinality) -1;
                }
                fitnessEvaluated = false;               // set this again since it
                fitness = rfitness = cfitness = 0;  // may be called more than once
        }

        public void clearChromosome() {
                for (int locus = 0; locus < Length; locus++)     {
                        gene[locus] = 0;
                }
                fitnessEvaluated = false;
                fitness = rfitness = cfitness = 0;
        }

        public int getGene(int locus) {
                return gene[locus];
        }

        public void setGene(int locus, int allele) {
                gene[locus] = allele;
                fitnessEvaluated = false;
                fitness = rfitness = cfitness = 0;
        }

        public void mutateGene(int locus) {
                // randomize this gene
                gene[locus] = MyRandom.integer(cardinality) -1;
                fitnessEvaluated = false;
                fitness = rfitness = cfitness = 0;
        }

        public String toGenotype() {
                String genotype = "";
                for (int locus = Length -1; locus >=0 ; locus--) {
                                genotype += String.valueOf(gene[locus]);
                }
                return genotype;
        }


        private double phenotype() {
                // evaluate chromosome as a number
                double value = 0;
                for (int locus = 0; locus < Length; locus++) {
                        value += Math.pow(cardinality, (double)locus )  * gene[locus];
                }
                return value;
        }

        public String toPhenotype() {
                return "x=" + phenotype();
        }

protected double evalChromosome() {
```

```
                String uniqId;
                double value;
                CallNativeGa p = new CallNativeGa();
                uniqId = "ga" + String.valueOf(idCount++);
                String chromosome = toGenotype();
                String input = p.runObjFunc(obFunction,chromosome,uniqId);
                Double dvalue = Double.valueOf(input);
                value = dvalue.doubleValue();
                return value;
        }
        static {
                System.loadLibrary("ganative");
        }

}
```

File Crossover.java

```
// Crossover: performs crossover of the two selected parents. */
public class Crossover {
        int cromLength,xoverType;
        boolean[] mask;
        boolean bit;
        int xoPoint1, xoPoint2;
        int locus,temp;

        public Crossover (int cl, int xot) {
                cromLength = cl;
                xoverType= xot;          // 0 = uniform, 1 = 1 point, 2 = 2 point
                mask = new boolean[cromLength];
                bit =true;
        }
        public void xOver(Chromosome one, Chromosome two) {
                // create xo mask
                xoPoint2 = MyRandom.integer(cromLength-1);
                if (xoverType == 1 ) {
                        // one point: start cut at zero
                        xoPoint1 =0;
                }
                else {
                        xoPoint1 = MyRandom.integer(cromLength-1);
                }
                if (xoverType == 0 ) {
                        for (locus = 0; locus < cromLength; locus++) {
                                mask[locus]= MyRandom.bool();
                        }
                }
                else {
                        if (xoPoint1 > xoPoint2 ) {
                                //reverse order and reverse bit flag;
                                bit = false;
                                temp =xoPoint2;
                                xoPoint2=xoPoint1;
                                xoPoint1=temp;
                        }
                        for (locus = 0; locus < cromLength; locus++)
                                mask[locus] = bit;
                        for (locus = xoPoint1; locus < xoPoint2; locus++)
                                mask[locus] = !bit;
                }
                // Do the Crossover using the mask
                        for (int locus = 0; locus < cromLength; locus++) {
                        if (mask[locus]) {
                                // swap
                                temp = one.getGene(locus);
                                one.setGene(locus, two.getGene(locus));
                                two.setGene(locus, temp);
                                //System.err.println("Got: " + one.toGenotype() + " and  " +
two.toGenotype());
                        }
                }
         }
}
```

File Selection.java

```
// Standard proportional selection using the roulette wheel method
public class Selection {
        public void select(Chromosome[] population, Chromosome[] newPopulation, int
populationSize){
                double p, sum = 0;
                int i;
                // find total fitness of the population
                for (i = 0; i < populationSize; i++) {
                        sum += population[i].getFitness();
                }

                // calculate relative fitness
                for (i = 0; i < populationSize; i++) {
                        population[i].rfitnessSet(population[i].getFitness()/sum);
                }

                population[0].cfitnessSet(population[0].rfitnessGet());

                // calculate cumulative fitness
                for (i = 1; i < populationSize; i++) {
                                population[i].cfitnessSet(population[i-1].cfitnessGet() +
population[i].rfitnessGet());
                }

                // finally select survivors using cumulative fitness.
                for (i = 0; i < populationSize; i++) {
                                p = MyRandom.dbl();
                                if (p < population[0].cfitnessGet())  {
                                        population[0].copyChromosome(newPopulation[i]);
                        }
                        else {
                                        for (int j = 0; j < populationSize; j++) {
                                                if (p >= population[j].cfitnessGet()
                                             && p < population[j+1].cfitnessGet()) {
                                             // note that population[populationSize-
1].cfitnessGet()
                                             //is 1.0, so j+1 never gets as big as populationSize

                                               population[j+1].copyChromosome(newPopulation[i]);
                                               int k = j+1;
                                        }
                                }
                                }
                }
        }
}
```

Java Native Method

File CallNativeGa.java

```
class CallNativeGa {
        // just define the interface, implementation is in loadable library
        public native String runObjFunc(String obfunc,
                String chromo, String uniqId);
}
```

File ganative.c

```
// Native Method Interface written in C
// Complied and loaded into Java as a library
#include <stdio.h>
#include <process.h>
#include <jni.h>
#include "CallNativeGa.h"

/* Native Method to Call Objective function */
JNIEXPORT jstring JNICALL Java_CallNativeGa_runObjFunc
                (JNIEnv *env, jobject obj, jstring obfunc, jstring chromo,
                        jstring uniqId ) {
        char command[128],returnbuf[128];
        FILE *f;
        int i;
        double value;
        //char value[128];
        const char *Cobfunc = (*env)->GetStringUTFChars(env, obfunc, 0);
        const char *Cchromo = (*env)->GetStringUTFChars(env, chromo, 0);
        const char *CuniqId = (*env)->GetStringUTFChars(env, uniqId, 0);

        if (strcmp(CuniqId,"decode") == 0 ) {
                sprintf(command,"%s %s",Cobfunc,Cchromo);
                i= system(command);
                sprintf(returnbuf,"%s","");
        }
        else {
                sprintf(command,"%s %s %s",Cobfunc,Cchromo,CuniqId);
                i= system(command);
                // Read results from a file created by the objective function
                if ((f = fopen(CuniqId, "rt")) == NULL) {
                        exit(3);
                }
                fscanf(f,"%lf",&value);
                sprintf(returnbuf,"%f",value);
                fclose(f);
                // remove the objective function return file
                unlink(CuniqId);
        }
        //Release java string environment
        (*env)->ReleaseStringUTFChars(env, obfunc, Cobfunc);
        (*env)->ReleaseStringUTFChars(env, chromo, Cchromo);
        (*env)->ReleaseStringUTFChars(env, uniqId, CuniqId);

        // return value as a string
        return (*env)->NewStringUTF(env, returnbuf);
}
```

File callGaNative.h

```
#include <jni.h>
/* Header for class CallNativeGa */

#ifndef _Included_CallNativeGa
#define _Included_CallNativeGa
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     CallNativeGa
 * Method:    runObjFunc
 * Signature: (Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_CallNativeGa_runObjFunc
  (JNIEnv *, jobject, jstring, jstring, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Vita Auctoris

Tom Williams obtained a Bachelor of Computer Science (Honours) degree from the **University of Windsor** in 1994. He is currently employed as a UNIX Systems Administrator and Webmaster and **St. Clair College**. He is a candidate for a Master's degree in Science at the University of Windsor and hopes to graduate in 1999.